

Kotlin 中文文档

目錄

analytics GA

2017.5.17

Android Announces Support for Kotlin

By Mike Cleron, Director, Android Platform

Google 宣布官方支持 Kotlin android-developers.googleblog

Android Studio 3.0 将默认集成 Kotlin plug-in ，博客中还说到 Kotlin 有着出色的设计，并相信 Kotlin 会帮助开发者更快更好的开发 Android 应用

Expedia, Flipboard, Pinterest, Square 等公司都有在自家的项目中使用 Kotlin

2017.3.8

Kotlin 1.1 正式发布，这次最令人振奋的莫过于协程的发布，有了协程就可以更优雅的完成异步编程了

更多新特性请参看what's new in kotlin 1.1

pdf下载 ePub下载

记得要点 star star star

发现有翻译的不好的或者错误欢迎到 github 提 issue

号外 号外 Kotlin 1.0 正式发布

Android 世界的 Swift 终于发布1.0版本

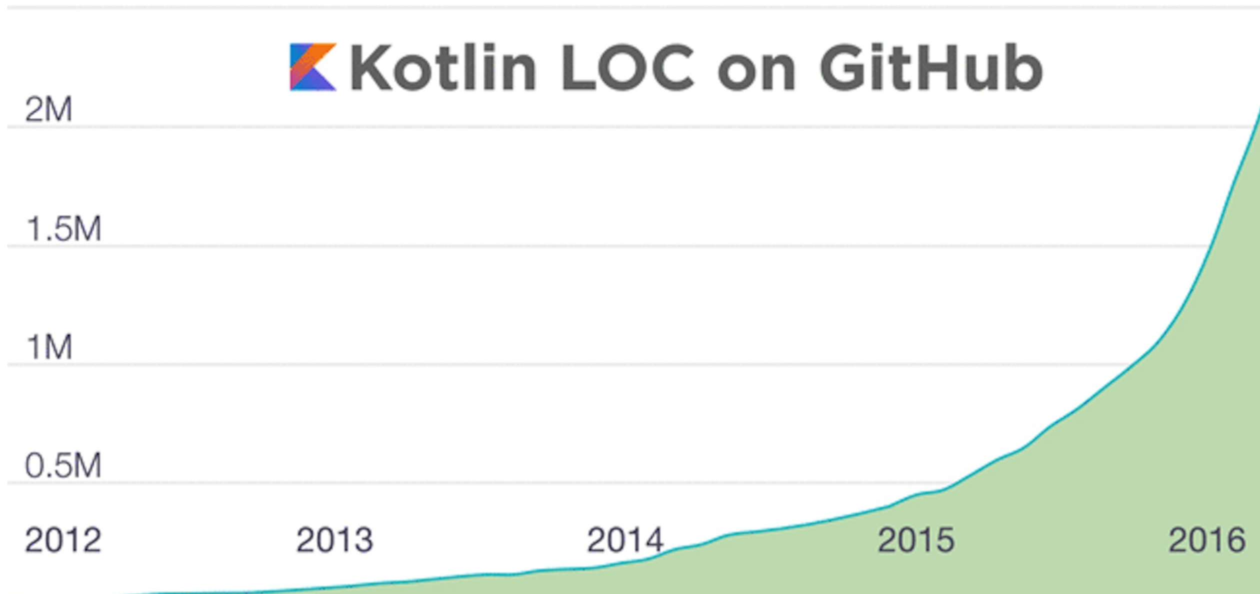
Kotlin 是一个实用性很强的语言，专注于互通，安全，简洁，工具健全...

无缝支持 Java+Kotlin 项目，可以更少的使用样版代码，确保类型安全。

Kotlin 1.0 更新日志

还换了logo :)

Kotlin LOC (软件规模代码行) 如下图



近期我会重新读一遍 Kotlin 官方文档 并对现在的这份文档进行更新(又立 flag 了) -- 2016.2.16

- 准备开始
  - 基本语法
  - 习惯用语
  - 编码风格
- 基础
  - 基本类型
  - 包
  - 控制流
  - 返回与跳转
- 类和对象
  - 类和继承
  - 属性和字段
  - 接口
  - 可见性修饰词
  - 扩展
  - 数据对象
  - 泛型
  - 嵌套类
  - 枚举类
  - 对象表达式和声明

- 基本语法
- 常用术语
- 编码风格
- 1.1中的新特性

原文

基本语法

包定义

在源文件的开头定义包：

```
package my.demo
import java.util.*
//...
```

包名不必和文件夹路径一致：源文件可以放在任意位置。

更多请参看 包(package)

定义函数

定义一个函数接受两个 int 型参数，返回值为 int ：

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

只有一个表达式作为函数体，以及自推导型的返回值：

```
fun sum(a: Int, b: Int) = a + b
```

返回一个没有意义的值：

```
fun printSum(a: Int, b: Int): Unit {
  println("sum of $a and $b is ${a + b}")
}
```

`Unit` 的返回类型可以省略：

```
fun printSum(a: Int, b: Int) {
  println("sum of $a and $b is ${a + b}")
}
```

更多请参看函数

定义局部变量

一次赋值（只读）的局部变量：

```
val a: Int = 1  // 立刻赋值
val b = 2   // `Int` 类型是自推导的
val c: Int  // 没有初始化器时要指定类型
c = 3       // 推导型赋值
```

可修改的变量：

```
var x = 5 // `Int` type is inferred
x += 1
```

更多请参看属性和字段

注释

与 java 和 javaScript 一样，Kotlin 支持单行注释和块注释。

```
// 单行注释

/*  哈哈哈哈
    这是块注释 */
```

与 java 不同的是 Kotlin 的 块注释可以级联。

参看文档化 Kotlin 代码学习更多关于文档化注释的语法。

使用字符串模板

```
var a = 1
// simple name in template:
val s1 = "a is $a"

a = 2
// arbitrary expression in template:
val s2 = "${s1.replace("is", "was")}, but now is $a"
```

更多请参看字符串模板

使用条件表达式

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

使用 if 作为表达式：

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

更多请参看 if 表达式

使用可空变量以及空值检查

当空值可能出现时应该明确指出该引用可空。

当 str 中不包含整数时返回空:

```
fun parseInt(str: String): Int? {
    // ...
}
```

使用一个返回可空值的函数：

```
fun parseInt(str: String): Int? {
  return str.toIntOrNull()
}

fun printProduct(arg1: String, arg2: String) {
  val x = parseInt(arg1)
  val y = parseInt(arg2)

  // 直接使用 x*y 会产生错误因为它们中有可能会有空值
  if (x != null && y != null) {
    // x 和 y 将会在空值检测后自动转换为非空值
    println(x * y)
  }
  else {
    println("either '$arg1' or '$arg2' is not a number")
  }
}
```

或者这样

```
if (x == null) {
    println("Wrong number format in arg1: '${arg1}'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '${arg2}'")
    return
}

// x and y are automatically cast to non-nullable after null che
ck
println(x * y)
```

更多请参看空安全

使用值检查以及自动转换

使用 is 操作符检查一个表达式是否是某个类型的实例。如果对不可变的局部变量或属性进行过了类型检查，就没有必要明确转换：

```
fun getStringLength(obj: Any): Int? {
  if (obj is String) {
    // obj 将会在这个分支中自动转换为 String 类型
    return obj.length
  }

  // obj 在类型检查分支外仍然是 Any 类型
  return null
}
```

或者这样

```
fun getStringLength(obj: Any): Int? {
  if (obj !is String) return null

  // obj 将会在这个分支中自动转换为 String 类型
  return obj.length
}
```

甚至可以这样

```
fun getStringLength(obj: Any): Int? {
    // obj 将会在&&右边自动转换为 String 类型
  if (obj is String && obj.length > 0) {
    return obj.length
  }

  return null
}
```

更多请参看 类 和 类型转换

使用 for 循环

```
val items = listOf("apple", "banana", "kiwi")
for (item in items) {
    println(item)
}
```

或者

```
val items = listOf("apple", "banana", "kiwi")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

参看for循环

使用 while 循环

```
val items = listOf("apple", "banana", "kiwi")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

参看while循环

使用 when 表达式

```
fun describe(obj: Any): String =
when (obj) {
    1          -> "One"
    "Hello"    -> "Greeting"
    is Long    -> "Long"
    !is String -> "Not a string"
    else       -> "Unknown"
}
```

参看when表达式

使用ranges

使用 in 操作符检查数值是否在某个范围内：

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

检查数值是否在范围外：

```
val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {
    println("-1 is out of range")
}
if (list.size !in list.indices) {
    println("list size is out of valid list indices range too")
}
```

使用范围内迭代：

```
for (x in 1..5) {
    print(x)
}
```

或者使用步进：

```
for (x in 1..10 step 2) {
    print(x)
}
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

参看Ranges

使用集合

对一个集合进行迭代：

```
for (item in items) {
    println(item)
}
```

使用 in 操作符检查集合中是否包含某个对象

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

使用lambda表达式过滤和映射集合：

```
fruits
.filter { it.startsWith("a") }
.sortedBy { it }
.map { it.toUpperCase() }
.forEach { println(it) }
```

参看高阶函数和lambda表达式

原文

习语

这里是一些在 Kotlin 中经常使用的习语。如果你有特别喜欢的习语想要贡献出来，赶快发起 pull request 吧。

创建DTOs(POJOs/POCOs) 数据类

```
data class Customer(val name: String, val email: String)
```

创建 Customer 类并带有如下方法：

> --为所有属性添加 getters ，如果为 var 类型同时添加 setters -- `equals()` -- `hashCode()` -- `toString()` -- `copy()` -- `component1()` , `component1()` , ... 参看数据类

函数默认参数

```
fun foo(a: Int = 0, b: String = "") { ... }
```

过滤 list

```
val positives = list.filter { x -> x > 0 }
```

或者更短：

```
val positives = list.filter { it > 0 }
```

字符串插值

```
println("Name $name")
```

实例检查

```
when (x) {
    is Foo ->  ...
    is Bar -> ...
    else -> ...
}
```

遍历 map/list 键值对

```
for ((k, v) in map) {
    print("$k -> $v")
}
```

k,v 可以随便命名

使用 ranges

```
for (i in 1..100) { ... }  // 闭区间: 包括100
for (i in 1 until 100) { ... } // 半开区间: 不包括100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

只读 list

```
val list = listOf("a", "b", "c")
```

只读map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

访问 map

```
println(map["key"])
map["key"] = value
```

懒属性(延迟加载)

```
val p: String by lazy {
    // compute the string
}
```

扩展函数

```
fun String.spcaceToCamelCase() { ... }
"Convert this to camelcase".spcaceToCamelCase()
```

创建单例模式

```
object Resource {
    val name = "Name"
}
```

如果不为空则... 的简写

```
val files = File("Test").listFiles()
println(files?.size)
```

如果不为空...否则... 的简写

```
val files = File("test").listFiles()
println(files?.size ?: "empty")
```

如果声明为空执行某操作

```
val data = ...
val email = data["email"] ?: throw IllegalStateException("Email
is missing!")
```

如果不为空执行某操作

```
val date = ...
data?.let{
    ...//如果不为空执行该语句块
}
```

如果不空则映射(Map nullable value if not null)

```
val data = ...

val mapped = data?.let { transformData(it) } ?: defaultValueIfDa
taIsNull
```

返回 when 判断

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color pa
ram value")
    }
}
```

try-catch 表达式

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // Working with result
}
```

if 表达式

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

使用生成器模式返回 Unit

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

单表达式函数

```
fun theAnswer() = 42
```

与下面的语句是等效的

```
fun theAnswer(): Int {
    return 42
}
```

可以和其它习语组合成高效简洁的代码。譬如说 when 表达式：

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param
value")
}
```

利用 with 调用一个对象实例的多个方法

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Java 7's try with resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

需要泛型类型信息的泛型函数的简便形式

```
//  public final class Gson {
//     ...
//     public <T> T fromJson(JsonElement json, Class<T> classOfT
) throws JsonSyntaxException {
//     ...

inline fun <reified T: Any> Gson.fromJson(json): T = this.fromJs
on(json, T::class.java)
```

消费一个可能为空的布尔值

```
val b: Boolean? = ...
if (b == true) {
    ...
} else {
    // `b` is false or null
}
```

原文

编码规范

本页包含了当前 kotlin 语言的代码风格。

命名风格

如有疑惑，默认为Java编码约定，比如：

> --使用骆驼命名法(在命名中避免下划线)
>
> --类型名称首字母大写
>
> --方法和属性首字母小写
>
> --缩进用四个空格
>
> --public 方法要写说明文档，这样它就可以出现在 Kotllin Doc 中

冒号

在冒号区分类型和父类型中要有空格，在实例和类型之间是没有空格的：

```
interface Foo<out T : Any> : Bar {
    fun foo(a: Int): T
}
```

Lambdas

在 Lambdas 表达式中，大括号与表达式间要有空格，箭头与参数和函数体间要有空格。lambda表达应尽可能不要写在圆括号中

```
list.filter { it > 10 }.map { element -> element * 2 }
```

在使用简短而非嵌套的lambda中，建议使用 `it` 而不是显式地声明参数。在使用参数的嵌套lambda中，参数应该总是显式声明

类声明格式

参数比较少的类可以用一行表示：

```
class Person(id: Int, name: String)
```

具有较多的参数的类应该格式化成每个构造函数的参数都位于与缩进的单独行中。此外，结束括号应该在新行上。如果我们使用继承，那么超类构造函数调用或实现的接口列表应该位于与括号相同的行中

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) {
    // ...
}
```

对于多个接口，应该首先定位超类构造函数调用，然后每个接口应该位于不同的行中

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker {
    // ...
}
```

构造函数参数可以使用常规缩进或连续缩进(双倍正常缩进)。

Unit

如果函数返回 Unit ，返回类型应该省略：

```
fun foo() { // ": Unit"被省略了
}
```

函数 vs 属性

在某些情况下，没有参数的函数可以与只读属性互换。尽管语义是相似的，但是有一些风格上的约定在什么时候更偏向于另一个。

在下面的情况下，更偏向于属性而不是一个函数:

> -- 不需要抛出异常 -- 复杂度为O(1) -- 低消耗的计算(或首次运行结果会被缓存)
> -- 返回与调用相同的结果

基本类型

在 Kotlin 中，所有的东西都是对象，这就意味着我们可以调用任何变量的成员函数和属性。一些类型是内建的，它们的实现是优化过的，但对用户来说它们就像普通的类一样。在这节中，我们将会讲到大多数的类型：数值，字符，布尔，以及数组。

数值

Kotlin 处理数值的方法和 java 很相似，但不是完全一样。比如，不存在隐式转换数值的精度，并且在字面上有一些小小的不同。

Kotlin 提供了如下内建数值类型(和 java 很相似)：

类型	位宽
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

注意字符在 Kotlin 中不是数值类型

字面值常量

主要是以下几种字面值常量：

> --十进制数值: `123`
> --长整型要加大写 `L` : `123L`
> --16进制：`0x0f`
> --二进制：`0b00001011`

注意不支持8进制

Kotlin 也支持传统的浮点数表示：

> -- 默认双精度浮点数(Double)：`123.5`，`123.5e10`
> -- 单精度浮点数(Float)要添加 `f` 或 `F` ：123.5f

数值常量中可以添加下划线分割(1.1版本新特性)

您可以使用下划线增加数值常量的可读性:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

表示

在 java 平台上，数值被 JVM 虚拟机以字节码的方式物理存储的，除非我们需要做可空标识(比如说 Int?) 或者涉及泛型。在后者中数值是装箱过的。

注意装箱过的数值是不保留特征的：

```
val a: Int = 10000
print (a === a ) // 打印 'true'
val boxedA: Int? =a
val anotherBoxedA: Int? = a
print (boxedA === anotherBoxedA ) // 注意这里打印的是 'false'
```

然而，它们是值相等的：

```
val a: Int = 10000
print(a == a) // 打印 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA == anotherBoxedA) // 打印 'true'
```

显式转换

由于不同的表示，低精度类型不是高精度类型的子类型。如果是的话我们就会碰到下面这样的麻烦了

```
// 这是些伪代码，不能编译的
val a: Int? =1 // 一个装箱过的 Int (java.lang.Integer)
val b: Long? = a // 一个隐式装箱的 Long (java.lang.Long)
print( a == b )// 很惊讶吧  这次打印出的是 'false' 这是由于 Long 类型的 equals() 只有和 Long 比较才会相同
```

因此不止是特征会丢失，有时候连值相等都会悄悄失效。

所以，低精度类型是不会隐式转换为高精度类型的。这意味着我们必须显式转换才能把 `Byte` 赋值给 `Int`

```
val b: Byte = 1 // OK, 字面值常量会被静态检查
val i: Int = b // ERROR
```

我们可以通过显式转换把数值类型提升

```
val i: Int = b.toInt() // 显式转换
```

每个数值类型都支持下面的转换：

> `toByte(): Byte`
>
> `toShort(): Short`
>
> `toInt(): Int`
>
> `toLong(): Long`
>
> `toFloat(): Float`
>
> `toDouble(): Double`
>
> `toChar(): Char`

隐式转换一般情况下是不容易被发觉的，因为我们使用了上下文推断出类型，并且算术运算会为合适的转换进行重载，比如

```
val l = 1.toLong + 1 // Long  + Int => Long
```

运算符

Kotlin支持标准的算术运算表达式，这些运算符被声明为相应类的成员(但是编译器将调用优化到相应的指令)。参看运算符重载。

至于位运算，Kotlin 并没有提供特殊的操作符，只是提供了命名函数，可以采用中缀形式调用，比如：

```
val x = (1 shl 2) and 0x000FF000
```

下面是全部的位运算操作符(只可以用在 `Int` 和 `Long` 类型)：

> `shl(bits)` – 有符号左移 (相当于 Java's `<<`) `shr(bits)` – 有符号右移 (相当于 Java's `>>`) `ushr(bits)` – 无符号右移 (相当于 Java's `>>>`) `and(bits)` – 按位与 `or(bits)` – 按位或 `xor(bits)` – 按位异或 `inv(bits)` – 按位翻转

字符

字符类型用 `Char` 表示。不能直接当做数值来使用

```
fun check(c: Char) {
    if (c == 1) { // ERROR: 类型不匹配
        // ...
    }
}
```

字符是由单引号包裹的：'1'，特殊的字符通过反斜杠\转义，下面的字符序列支持转义： `\t` , `\b` , `\n` , `\r` , `\'` , `\"` , `\\` 和 `\$` 。编码任何其他字符，使用Unicode 转义语法： `\uFF00` 。

我们可以将字符显示的转义为Int数字：

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() //显示转换为数值类型
}
```

和数值类型一样，需要一个可空引用时，字符会被装箱。特性不会被装箱保留。

布尔值

布尔值只有 true 或者 false

如果需要一个可空引用，将会对布尔值装箱

布尔值的内建操作包括

> `||` – 短路或
>
> `&&` – 短路与
>
> `!` - 取反

数组

数组在 Kotlin 中由 `Array` 类表示，有 `get` 和 `set` （通过运算符重载为 `[]` ）方法，和 `size` 属性，以及一些常用的函数：

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
  operator fun set(index: Int, value: T): Unit

  operator fun iterator(): Iterator<T>
  // ...
}
```

给库函数 `arrayOf()` 传递每一项的值来创建Array，`arrayOf(1, 2, 3)` 创建了一个[1, 2, 3] 这样的数组。也可以使用库函数 `arrayOfNulls()` 创建一个指定大小的空Array。

另一种方式就是使用工厂函数，接受一个数组大小参数以及一个可以根据给定索引创建初始值的函数：

```
// 创建一个 Array<String>  内容为 ["0", "1", "4", "9", "16"]
val asc = Array(5, {i -> (i * i).toString() })
```

像我们上面提到的，`[]` 操作符表示调用 `get()` `set()` 函数

注意：和 java 不一样，arrays 在 kotlin 中是不可变的。这意味这 kotlin 不允许我们把 `Array<String>` 转为 `Array<Any>` ,这样就阻止了可能的运行时错误(但你可以使用 `Array<outAny>` , 参看 Type Projections)

Kotlin 有专门的类来表示原始类型从而避免过度装箱： ByteArray, ShortArray, IntArray 等等。这些类与 Array 没有继承关系，但它们有一样的方法与属性。每个都有对应的库函数：

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

字符串

字符串是由 `String` 表示的。字符串是不变的。字符串的元素可以通过索引操作读取: `s[i]` 。字符串可以用 for 循环迭代：

```
for (c in str) {
    println(c)
}
```

字符串字面值

Kotlin 有两种类型的字符串字面值：一种是可以带转义符的，一种是可以包含新行以及任意文本的。带转义符的 string 很像 java 的 string:

```
val s = "Hello World!\n"
```

转义是使用传统的反斜线的方式。参见Characters，查看支持的转义序列。

整行String 是由三个引号包裹的(`"""`),不可以包含转义符但可 以包含其它字符：

```
val text = """
    for (c in "foo")
        print(c)
"""
```

你可以通过 trim-margin() 函数移除空格：

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

默认采用 `|` 标注起始前缀，也可以传递其它的字符做起始前缀，比如 `trimMargin(">")`

字符串模板

字符串可以包含模板表达式，即可求值的代码片段，并将其结果连接到字符串中。模板表达式由 $ 开始并包含另一个简单的名称：

```
val i = 10
val s = "i = $i" // 求值为 "i = 10"
```

或者是一个带大括号的表达式：

```
val s = "abc"
val str = "$s.length is ${s.length}" // 结果为 "abc.length is 3"
```

模板既可以原始字符串中使用，也可以在转义字符串中使用。如果需要在原始字符串(不支持反斜杠转义)中表示一个文字$字符，那么可以使用以下语法：

```
val price = """
${'$'}9.99
"""
```

包

代码文件以包声明开始：

```
package foo.bar

fun bza() {}

class Goo {}

//...
```

代码文件的所有内容(比如类和函数)都被包含在包声明中。因此在上面的例子中， `bza()` 的全名应该是 `foo.bar.bza` ， `Goo` 的全名是 `foo.bar.Goo` 。

如果没有指定包名，那这个文件的内容就从属于没有名字的 "default" 包。

默认导入

许多包被默认导入到每个Kotlin文件中：

> -- kotlin.*
>
> -- kotlin.annotation.*
>
> -- kotlin.collections.*
>
> -- kotlin.comparisons.* (since 1.1)
>
> -- kotlin.io.*
>
> -- kotlin.ranges.*
>
> -- kotlin.sequences.*
>
> -- kotlin.text.*

一些附加包会根据平台来决定是否默认导入：

> -- JVM:
>
> ---- java.lang.*
>
> ---- kotlin.jvm.*
>
> -- JS:
>
> ---- kotlin.js.*

Imports

除了模块中默认导入的包，每个文件都可以有导入自己需要的包。导入语法可以在 grammar 查看。

我们可以导入一个单独的名字，比如下面这样：

```
import foo.Bar // Bar 现在可以直接使用了
```

或者范围内的所有可用的内容 (包，类，对象，等等):

```
import foo.*/ /foo 中的所有都可以使用
```

如果命名有冲突，我们可以使用 `as` 关键字局部重命名解决冲突

```
import foo.Bar // Bar 可以使用
import bar.Bar as bBar // bBar 代表 'bar.Bar'
```

import关键字不局限于导入类;您也可以使用它来导入其他声明:

> -- 顶级函数与属性
>
> -- 在对象声明中声明的函数和属性
>
> -- 枚举常量

与 Java 不同的是，Koting 没有静态导入的语法，所有的导入都是通过 `import` 关键字声明的。

顶级声明的可见性

如果最顶的声明标注为 private , 那么它是声明文件私有的 (参看 Visibility Modifiers)。

流程控制：if , when , for , while

if 表达式

在 Kotlin 中，if 是带有返回值的表达式。因此Kotlin没有三元运算符(condition ? then : else),因为 if 语句可以做到同样的事。

```
// 传统用法
var max = a
if (a < b) max = b

// 带 else
var max: Int
if (a > b) {
    max = a
}
else{
    max = b
}
// 作为表达式
val max = if (a > b) a else b
```

if分支可以作为块，最后一个表达式是该块的值：

```
val max = if (a > b){
    print("Choose a")
    a
}
else{
    print("Choose b")
    b
}
```

如果使用If作为表达式而不是声明(例如，返回其值或将其赋值给一个变量)，表达式必须带有 else 分支。

参看 if语法

When 表达式

when 取代了 C 风格语言的 switch 。最简单的用法像下面这样

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

when会对所有的分支进行检查直到有一个条件满足。when 可以用做表达式或声明。如果用作表达式的话，那么满足条件的分支就是表达式的值jjs。如果用做声明，那么分支的值会被忽略。(与 if 表达式一样，每个分支是一个语句块，而且它的值就是最后一个表达式的值)

在其它分支都不匹配的时候默认匹配 else 分支。如果把 when 做为表达式的话 else 分支是强制的，除非编译器可以证明分支条件已经覆盖所有可能性。

如果有分支可以用同样的方式处理的话，分支条件可以连在一起：

```
when (x) {
    0,1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

可以用任意表达式作为分支的条件

```
when (x) {
    parseInt(s) -> print("s encode x")
    else -> print("s does not encode x")
}
```

甚至可以用 in 或者 !in 检查值是否值在一个范围或一个集合中：

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

也可以用 is 或者 !is 来判断值是否是某个类型。注意，由于 smart casts ，可以不进行类型检查就可以使用相应的属性或方法。

```
val hasPrefix = when (x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

when 也可以用来代替 if-else if 。如果没有任何参数提供，那么分支的条件就是简单的布尔表达式，当条件为真时执行相应的分支：

```
when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

参看when语法

for 循环

for 循环可以对所有提供迭代器的变量进行迭代。等同于 C# 等语言中的 `foreach` 。语法形式：

```
for (item in collection)
    print(item)
```

内容可以是一个语句块

```
for (item: Int in ints){
    // ...
}
```

像之前提到的， for 可以对任何提供的迭代器的变量进行迭代，比如：

> -- 有一个 iterator() 成员函数或扩展函数，其返回类型要
>
> -- 有一个 next() 成员函数或扩展函数, 并且
>
> -- 有一个返回 Boolean 的 hasNext() 成员函数或扩展函数。

这三个函数都需要标记为运算符.

对数组的for循环不会创建迭代器对象，而是被编译成一个基于索引的循环.

如果你想通过 list 或者 array 的索引进行迭代，你可以这样做：

```
for (i in array.indices)
    print(array[i])
```

这里需要说明的是 "iteration through a range " 会被自动编译成最优的实现，并不会有附加对象生成。

或者，您也可以使用 `withIndex` 库函数

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

参看for语法

while 循环

while 和 do...while 和其它语言没什么区别

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y 在这是可见的
```

参看while 语法

在循环中使用 break 和 continue

kotlin 支持传统的 break 和 continue 操作符。参看返回和跳转

返回与跳转

Kotlin 有三种结构跳转表达式：

> -- return 默认情况下从最近的闭合函数或者匿名函数返回。 -- break 结束最近的闭合循环
> -- continue 跳到最近的闭合循环的下一次循环

上述表达式都可以作为更大的表达式的一部分：

```
val s = person.name ?: return
```

这些表达式的类型是 Nothing type

break 和 continue 标签

Kotlin 中任意表达式可以添加标签。标签通过 @ 结尾来标识，比如： `abc@` ， `fooBar@` 都是有效的(参看语法)。使用标签表达式，只需像这样：

```
loop@ for (i in 1..100){
    // ...
}
```

我们可以用标签实现 break 或者 continue 的快速跳转：

```
loop@ for (i in 1..100) {
    for (j in i..100) {
        if (...)
            break@loop
    }
}
```

break 是跳转标签后面的表达式，continue 是跳转到循环的下一次迭代。

返回到标签

在字面函数，局部函数，以及对象表达式中，函数在Kotlin中是可以嵌套的。合法的return 允许我们返回到外层函数。最重要的使用场景就是从lambda表达式中返回，还记得我们之前的写法吗：

```
fun foo() {
    ints.forEach {
        if (it  == 0) return
        print(it)
    }
}
```

return 表达式返回到最近的闭合函数，比如 `foo` (注意这样非局部返回仅仅可以在内联函数中使用)。如果我们需要从一个字面函数返回可以使用标签修饰 return :

```
fun foo() {
    ints.forEach lit@ {
        if (it ==0) return@lit
        print(it)
    }
}
```

现在它仅仅从字面函数中返回。经常用一种更方便的含蓄的标签：比如用和传入的 lambda 表达式名字相同的标签。

```
fun foo() {
    ints.forEach {
        if (it == 0) return@forEach
        print(it)
    }
}
```

另外，我们可以用函数表达式替代匿名函数。在函数表达式中使用 return 语句可以从函数表达式中返回。

```
fun foo() {
    ints.forEach(fun(value:  Int){
        if (value == 0) return
        print(value)
    })
}
```

当返回一个值时，解析器给了一个参考，比如(原文When returning a value, the parser gives preference to the qualified return, i.e.)：

```
return@a 1
```

表示“在标签 `@a` 返回 `1` ”而不是返回一个标签表达式 `(@a 1)`

命名函数自动定义标签：

```
foo outer() {
    foo inner() {
        return@outer
    }
}
```

类和继承

类

在 Kotlin 中类用 `class` 时：

```
class Invoice {
}
```

类的声明包含类名，类头(指定类型参数，主构造函数等等)，以及类主体，用大括号包裹。类头和类体是可选的；如果没有类体可以省略大括号。

```
class Empty
```

构造函数

在 Kotlin 中类可以有一个主构造函数以及多个二级构造函数。主构造函数是类头的一部分：跟在类名后面(可以有可选的类型参数)。

```
class Person constructor(firstName: String) {
}
```

如果主构造函数没有注解或可见性说明，则 `constructor` 关键字是可以省略：

```
class Person(firstName: String){
}
```

主构造函数不能包含任意代码。初始化代码可以放在以 `init` 做前缀的初始化块内

```
class Customer(name: String) {
    init {
        logger,info("Customer initialized with value ${name}")
    }
}
```

注意主构造函数的参数可以用在初始化块内，也可以用在类的属性初始化声明处：

```
class Customer(name: String) {
    val customerKry = name.toUpperCase()
}
```

事实上，声明属性并在主构造函数中初始化,在 Kotlin 中有更简单的语法：

```
class Person(val firstName: String, val lastName: String, var age
: Int) {
}
```

就像普通的属性，在主构造函数中的属性可以是可变的(`var`)或只读的(`val`)。

如果构造函数有注解或可见性声明，则 `constructor` 关键字是不可少的，并且可见性应该在前：

```
class Customer public @inject constructor (name: String) {...}
```

参看可见性

二级构造函数

类也可以有二级构造函数，需要加前缀 `constructor` :

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

如果类有主构造函数，每个二级构造函数都要，或直接或间接通过另一个二级构造函数代理主构造函数。在同一个类中代理另一个构造函数使用 `this` 关键字：

```
class Person(val name: String) {
    constructor (name: String, paret: Person) : this(name) {
        parent.children.add(this)
    }
}
```

如果一个非抽象类没有声明构造函数(主构造函数或二级构造函数)，它会产生一个没有参数的构造函数。该构造函数的可见性是 public 。如果你不想你的类有公共的构造函数，你就得声明一个拥有非默认可见性的空主构造函数：

```
class DontCreateMe private constructor () {
}
```

> 注意：在 JVM 虚拟机中，如果主构造函数的所有参数都有默认值，编译器会生成一个附加的无参的构造函数，这个构造函数会直接使用默认值。这使得 Kotlin 可以更简单的使用像 Jackson 或者 JPA 这样使用无参构造函数来创建类实例的库。

```
class Customer(val customerName: String = "")
```

创建类的实例

我们可以像使用普通函数那样使用构造函数创建类实例：

```
val invoice = Invoice()
val customer = Customer("Joe Smith")
```

注意 Kotlin 没有 `new` 关键字。

创建嵌套类、内部类或匿名类的实例参见嵌套类

类成员

类可以包含：

> -- 构造函数和初始化代码块
>
> -- 函数
>
> -- 属性
>
> -- 内部类
>
> -- 对象声明

继承

Kotlin 中所有的类都有共同的父类 `Any` ，它是一个没有父类声明的类的默认父类：

```
class Example // 隐式继承于 Any
```

`Any` 不是 `java.lang.Object` ；事实上它除了 `equals()` , `hashCode()` 以及 `toString()` 外没有任何成员了。参看Java interoperability了解更多详情。

声明一个明确的父类，需要在类头后加冒号再加父类：

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

如果类有主构造函数，则基类可以而且是必须在主构造函数中使用参数立即初始化。

如果类没有主构造函数，则必须在每一个构造函数中用 `super` 关键字初始化基类，或者在代理另一个构造函数做这件事。注意在这种情形中不同的二级构造函数可以调用基类不同的构造方法：

```
class MyView : View {
    constructor(ctx: Context) : super(ctx) {
    }
    constructor(ctx: Context, attrs: AttributeSet) : super(ctx,a
ttrs) {
    }
}
```

`open` 注解与java中的 `final` 相反:它允许别的类继承这个类。默认情形下，kotlin中所有的类都是 final ,对应 Effective Java ：Design and document for inheritance or else prohibit it.

复写方法

像之前提到的，我们在 kotlin 中坚持做明确的事。不像 java ，kotlin 需要把可以复写的成员都明确注解出来，并且重写它们：

```
open class Base {
    open fun v() {}
    fun nv() {}
}

class Derived() : Base() {
    override fun v() {}
}
```

对于 `Derived.v()` 来说 `override` 注解是必须的。如果没有加的话，编译器会提示。如果没有 `open` 注解，像 `Base.nv()` ,在子类中声明一个同样的函数是不合法的，要么加 `override` 要么不要复写。在 final 类(就是没有open注解的类)中， `open` 类型的成员是不允许的。

标记为 `override` 的成员是open的，它可以在子类中被复写。如果你不想被重写就要加 final:

```
open class AnotherDerived() : Base() {
    final override fun v() {}
}
```

等等！我现在怎么**hack**我的库？！

有个问题就是如何复写子类中那些作者不想被重写的类，下面介绍一些令人讨厌的方案。

我们认为这是不好的，原因如下：

> 最好的实践建议你不应给做这些 hack
>
> 人们可以用其他的语言成功做到类似的事情
>
> 如果你真的想 hack 那么你可以在 java 中写好 hack 方案，然后在 kotlin 中调用 (参看java调用)，专业的构架可以很好的做到这一点

复写属性

复写属性与复写方法类似，在一个父类上声明的属性在子类上被重新声明，必须添加 `override` ，并且它们必须具有兼容的类型。每个被声明的属性都可以被一个带有初始化器的属性或带有getter方法的属性覆盖

```
open class Foo {
  open val x: Int get { ... }
}

class Bar1 : Foo() {
  override val x: Int = ...
}
```

您还可以使用 `var` 属性覆盖一个 `val` 属性，但反之则不允许。这是允许的，因为 `val` 属性本质上声明了一个getter方法，并将其重写为 `var` ，另外在派生类中声明了setter方法。

注意，可以在主构造函数中使用 `override` 关键字作为属性声明的一部分。

```
interface Foo {
    val count: Int
}

class Bar1(override val count: Int) : Foo

class Bar2 : Foo {
    override var count: Int = 0
}
```

复写规则

在 kotlin 中，实现继承通常遵循如下规则：如果一个类从它的直接父类继承了同一个成员的多个实现，那么它必须复写这个成员并且提供自己的实现(或许只是直接用了继承来的实现)。为表示使用父类中提供的方法我们用 `super<Base>` 表示:

```
open class A {
    open fun f () { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // 接口的成员变量默认是 open 的
    fun b() { print("b") }
}

class C() : A() , B {
    // 编译器会要求复写f()
    override fun f() {
        super<A>.f() // 调用 A.f()
        super<B>.f() // 调用 B.f()
    }
}
```

可以同时从 A 和 B 中继承方法，而且 C 继承 a() 或 b() 的实现没有任何问题，因为它们都只有一个实现。但是 f() 有俩个实现，因此我们在 C 中必须复写 f() 并且提供自己的实现来消除歧义。

抽象类

一个类或一些成员可能被声明成 abstract 。一个抽象方法在它的类中没有实现方法。记住我们不用给一个抽象类或函数添加 open 注解，它默认是带着的。

我们可以用一个抽象成员去复写一个带 open 注解的非抽象方法。

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base() {
    override abstract fun f()
}
```

伴随对象

在 kotlin 中不像 java 或者 C# 它没有静态方法。在大多数情形下，我们建议只用包级别的函数。

如果你要写一个没有实例类就可以调用的方法，但需要访问到类内部(比如说一个工厂方法)，你可以把它写成它所在类的一个成员(you can write it as a member of an object declaration inside that class)

更高效的方法是，你可以在你的类中声明一个伴随对象，这样你就可以像 java/c# 那样把它当做静态方法调用，只需要它的类名做一个识别就好了

密封类

密封类用于代表严格的类结构，值只能是有限集合中的某种类型，不可以是任何其它类型。这就相当于一个枚举类的扩展：枚举值集合的类型是严格限制的，但每个枚举常量只有一个实例，而密封类的子类可以有包含不同状态的多个实例。

声明密封类需要在 class 前加一个 sealed 修饰符。密封类可以有子类但必须全部嵌套在密封类声明内部、

```
sealed class Expr {
    class Const(val number: Double) : Expr()
    class Sum(val e1: Expr, val e2: Expr) : Expr()
    object NotANumber : Expr()
}
```

注意密封类子类的扩展可以在任何地方，不必在密封类声明内部进行。

使用密封类的最主要的的好处体现在你使用 when 表达式。可以确保声明可以覆盖到所有的情形，不需要再使用 else 情形。

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered a
ll the cases
}
```

属性和字段

属性声明

在 Kotlin 中类可以有属性，我们可以使用 var 关键字声明可变属性，或者用 val 关键字声明只读属性。

```
class Address {
    var name: String = ...
  var street: String = ...
    var city: String = ...
  var state: String? = ...
    var zip: String = ...
}
```

我们可以像使用 java 中的字段那样,通过名字直接使用一个属性：

```
fun copyAddress(address: Address) : Address {
    val result = Address() // 在 kotlin 中没有 new 关键字
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

Getters 和 Setters

声明一个属性的完整语法如下：

```
var <propertyName>: <PropertyType> [ = <property_initializer> ]
    <getter>
    <setter>
```

语法中的初始化语句，getter 和 setter 都是可选的。如果属性类型可以从初始化语句或者类的成员函数中推断出来,那么他的类型也是忽略的。

例子：

```
var allByDefault: Int? // 错误: 需要一个初始化语句, 默认实现了 getter 和 setter 方法
var initialized = 1 // 类型为 Int, 默认实现了 getter 和 setter
```

只读属性的声明语法和可变属性的声明语法相比有两点不同: 它以 `val` 而不是 `var` 开头，不允许 setter 函数：

```
val simple: Int? // 类型为 Int ，默认实现 getter ，但必须在构造函数中初始化

val inferredType = 1 // 类型为 Int 类型,默认实现 getter
```

我们可以像写普通函数那样在属性声明中自定义的访问器，下面是一个自定义的 getter 的例子:

```
var isEmpty: Boolean
    get() = this.size == 0
```

下面是一个自定义的setter:

```
var stringRepresentation: String
    get() = this.toString()
    set (value) {
        setDataFormString(value) // 格式化字符串,并且将值重新赋值给其他元素
}
```

为了方便起见,setter 方法的参数名是 `value` ,你也可以自己任选一个自己喜欢的名称.

从Kotlin 1.1开始，如果可以从getter方法推断出类型则可以省略之：

```
val isEmpty get() = this.size == 0  //  拥有Boolean类型
```

如果你需要改变一个访问器的可见性或者给它添加注解，但又不想改变默认的实现，那么你可以定义一个不带函数体的访问器:

```
var setterVisibility: String = "abc"// 非空类型必须初始化
    private set // setter是私有的并且有默认的实现
var setterWithAnnotation: Any?
    @Inject set // 用 Inject 注解 setter
```

备用字段

在 kotlin 中类不可以有字段。然而当使用自定义的访问器时有时候需要备用字段。出于这些原因 kotlin 使用 `field` 关键词提供了自动备用字段，

```
var counter = 0 // 初始化值会直接写入备用字段
    set(value) {
        if (value >= 0)
            field  = value
    }
```

`field` 关键词只能用于属性的访问器.

编译器会检查访问器的代码,如果使用了备用字段(或者访问器是默认的实现逻辑)，就会自动生成备用字段,否则就不会.

比如下面的例子中就不会有备用字段：

```
val isEmpty: Boolean
    get() = this.size == 0
```

备用属性

如果你想要做一些事情但不适合这种 "隐含备用字段" 方案，你可以试着用备用属性的方式：

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
    if (_table == null)
        _table = HashMap() //参数类型是推导出来的
        return _table ?: throw AssertionError("Set to null by an
other thread")
    }
```

综合来讲，这些和 java 很相似，可以避免函数访问私有属性而破坏它的结构

编译时常量

那些在编译时就能知道具体值的属性可以使用 `const` 修饰符标记为 编译时常量. 这种属性需要同时满足以下条件:

- 在top-level声明的 或者 是一个 `object` 的成员(Top-level or member of an object)
- 以 `String` 或基本类型进行初始化
- 没有自定义getter

这种属性可以被当做注解使用:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is depr
ecated"
@Deprected(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

延迟初始化属性

通常,那些被定义为拥有非空类型的属性,都需要在构造器中初始化.但有时候这并没有那么方便.例如在单元测试中,属性应该通过依赖注入进行初始化, 或者通过一个 setup 方法进行初始化.在这种条件下,你不能在构造器中提供一个非空的初始化语句,但是你仍然希望在访问这个属性的时候,避免非空检查.

为了处理这种情况,你可以为这个属性加上 `lateinit` 修饰符

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()
    }
}
```

这个修饰符只能够被用在类的 `var` 类型的可变属性定义中,不能用在构造方法中.并且属性不能有自定义的 getter 和 setter访问器.这个属性的类型必须是非空的,同样也不能为一个基本类型.

在一个 `lateinit` 的属性初始化前访问他,会导致一个特定异常,告诉你访问的时候值还没有初始化.

复写属性

参看复写成员

代理属性

最常见的属性就是从备用属性中读（或者写）。另一方面，自定义的 getter 和 setter 可以实现属性的任何操作。有些像懒值(lazy values)，根据给定的关键字从 map 中读出，读取数据库，通知一个监听者等等，像这些操作介于 getter setter 模式之间。

像这样常用操作可以通过代理属性作为库来实现。更多请参看这里。

接口

Kotlin 的接口很像 java 8。它们都可以包含抽象方法，以及方法的实现。和抽象类不同的是，接口不能保存状态。可以有属性但必须是抽象的，或者提供访问器的实现。

接口用关键字 `interface` 来定义：

```
interface MyInterface {
    fun bar()
    fun foo() {
        //函数体是可选的
    }
}
```

实现接口

一个类或对象可以实现一个或多个接口

```
class Child : MyInterface {
    fun bar () {
        //函数体
    }
}
```

接口中的属性

可以在接口中申明属性。接口中的属性要么是抽象的，要么提供访问器的实现。接口属性不可以有后备字段。而且访问器不可以引用它们。

```
interface MyInterface {
    val property: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

解决重写冲突

当我们在父类中声明了许多类型，有可能出现一个方法的多种实现。比如：

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

A B 接口都有声明了 foo() 和 bar() 函数。它们都实现了 foo() 方法，但只有 B 实现了 bar() ,bar() 在 A 中并没有声明它是抽象的，这是因为在接口中如果函数没有函数体，那么默认是抽像的。

不过，如果我们从 A 中派生一个 C 实体类，显然我们需要重写 bar() ，并实现它。而我们从 A 和 B 派生一个 D ，我们不用重写 bar() 方法，因为我们的一个继承中有一个已经实现了它。但我们继承了两个 foo() 的实现，因此编译器不知道应该选哪个，并强制我们重写 foo() 并且明确指出我们想怎么实现。

可见性修饰词

类，对象，接口，构造函数，属性以及它们的 setter 方法都可以有可见性修饰词。(getter与对应的属性拥有相同的可见性)。在 Kotlin 中有四种修饰词： `private` , `protected` , `internal` ,以及 `public` 。默认的修饰符是 `public` 。 下面将解释不同类型的声明作用域。

包

函数，属性和类，对象和接口可以在 "top-level" 声明，即可以直接属于包：

```
// 文件名: example.kt
package foo

fun baz() {}
class bar {}
```

> -- 如果没有指明任何可见性修饰词，默认使用 `public` ,这意味着你的声明在任何地方都可见；
>
> -- 如果你声明为 `private` ，则只在包含声明的文件中可见；
>
> -- 如果用 `internal` 声明，则在同一模块中的任何地方可见；
>
> -- `protected` 在 "top-level" 中不可以使用

例子：

```
// 文件名: example.kt
package foo

private fun foo() {} // 在example.kt可见

public var bar: Int = 5 // 属性在认可地方都可见
    private set // setter仅在example.kt中可见

internal val baz = 6 // 在同一module中可见
```

类和接口

当在类中声明成员时：

> `private` 只在该类(以及它的成员)中可见
>
> `protected` 和 `private` 一样但在子类中也可见
>
> `internal` 在本模块的所有可以访问到声明区域的均可以访问该类的所有 `internal` 成员 (internal — any client inside this module who sees the declaring class sees its internal members;)
>
> `public` 任何地方可见 (public — any client who sees the declaring class sees its public members.)

java 使用者注意：外部类不可以访问内部类的 private 成员。

如果你复写了一个 `protected` 成员并且没有指定可见性，那么该复写的成员具有 `protected` 可见性

例子：

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4 // 默认是public

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a不可见
    // b,c和d是可见的
    // 内部类和e都是可见的

    override val b = 5   // 'b' i具有protected可见性
}

class Unrelated(o: Outer) {
    // o.a, o.b 不可见
    // o.c and o.d 可见 (same module)
    // Outer.Nested 不可见, and Nested::e 也不可见
}
```

构造函数

通过下面的语法来指定主构造函数(必须显示的使用 constructor 关键字)的可见性：

```
class C private constructor(a: Int) { ... }
```

这里构造函数是 private 。所有的构造函数默认是 `public` ,实际上只要类是可见的它们就是可见的 (注意 `internal` 类型的类中的 public 属性只能在同一个模块内才可以访问)

局部声明

局部变量，函数和类是不允许使用修饰词的

模块

`internal` 修饰符是指成员的可见性是只在同一个模块中才可见的。模块在 Kotlin 中就是一系列的 Kotlin 文件编译在一起：

— an IntelliJ IDEA module;

— a Maven or Gradle project;

— a set of files compiled with one invocation of the Ant task.

扩展

与 C# 和 Gosu 类似, Kotlin 也提供了一种,可以在不继承父类，也不使用类似装饰器这样的设计模式的情况下对指定类进行扩展。我们可以通过一种叫做扩展的特殊声明来实现他。Kotlin 支持函数扩展和属性扩展。

函数扩展

为了声明一个函数扩展，我们需要在函数前加一个接收者类型作为前缀。下面我们会为 `MutableList<Int>` 添加一个 `swap` 函数：

```
fun MutableList<Int>.swap(x: Int, y: Int) {
    val tmp = this[x] // this 对应 list
    this[x] = this[y]
    this[y] = tmp
}
```

在扩展函数中的 this 关键字对应接收者对象。现在我们可以在任何 `MutableList<Int>` 实例中使用这个函数了：

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2)// 在 `swap()` 函数中 `this` 持有的值是 `l`
```

当然，这个函数对任意的 `MutableList<T>` 都是适用的，而且我们可以把它变的通用：

```
fun <T> MutableList<T>.swap(x: Int, y: Int) {
  val tmp = this[x] // 'this' corresponds to the list
  this[x] = this[y]
  this[y] = tmp
}
```

我们在函数名前声明了通用类型，从而使它可以接受任何参数。参看泛型函数。

扩展是被静态解析的

扩展实际上并没有修改它所扩展的类。定义一个扩展，你并没有在类中插入一个新的成员，只是让这个类的实例对象能够通过 `.` 调用新的函数。

需要强调的是扩展函数是静态分发的，举个例子,它们并不是接受者类型的虚拟方法。这意味着扩展函数的调用是由发起函数调用的表达式的类型决定的，而不是在运行时动态获得的表达式的类型决定。比如

```
open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())
```

这个例子会输出 `c` ，因为这里扩展函数的调用决定于声明的参数 `c` 的类型，也就是 `C` 。

如果有同名同参数的成员函数和扩展函数，调用的时候必然会使用成员函数，比如：

```

class C {
    fun foo() { println("member") }

}
fun C.foo() { println("extension") }
```

当我们对C的实例c调用 `c.foo()` 的时候,他会输出"member",而不是"extension"

但你可以用不同的函数签名通过扩展函数的方式重载函数的成员函数，比如下面这样：

```
class C {
    fun foo() { println("number") }
}

fun C.foo(i:Int) { println("extention") }
```

`C().foo(1)` 的调用会打印 “extentions”。

可空的接收者

注意扩展可以使用空接收者类型进行定义。这样的扩展使得，即使是一个空对象仍然可以调用该扩展，然后在扩展的内部进行 `this == null` 的判断。这样你就可以在 Kotlin 中任意调用 toString() 方法而不进行空指针检查：空指针检查延后到扩展函数中完成。

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // 在空检查之后，`this` 被自动转为非空类型，因此 toString() 可以被解析到任何类的成员函数中
    return toString()
}
```

属性扩展

和函数类似， Kotlin 也支持属性扩展：

```
val <T> List<T>.lastIndex:  Int
    get() = size-1
```

注意，由于扩展并不会真正给类添加了成员属性，因此也没有办法让扩展属性拥有一个备份字段.这也是为什么初始化函数不允许有扩展属性。扩展属性只能够通过明确提供 getter 和 setter方法来进行定义.

例子：

```
val Foo.bar = 1 //error: initializers are not allowed for extens
ion properties
```

伴随对象扩展

如果一个对象定义了伴随对象，你也可以给伴随对象添加扩展函数或扩展属性：

```
class MyClass {
    companion object {}
}
fun MyClass.Companion.foo(){

}
```

和普通伴随对象的成员一样，它们可以只用类的名字就调用：

```
MyClass.foo()
```

扩展的域

大多数时候我们在 top level 定义扩展，就在包下面直接定义：

```
package foo.bar
fun Baz.goo() { ... }
```

为了在除声明的包外使用这个扩展，我们需要在 import 时导入：

```
package com.example.usage

import foo.bar.goo // 导入所有名字叫 "goo" 的扩展

                   // 或者

import foo.bar.* // 导入foo.bar包下得所有数据

fun usage(baz: Baz) {
    baz.goo()
}
```

动机

在 java 中，我们通常使用一系列名字为 "*Utils" 的类: `FileUtils`，`StringUtils` 等等。很有名的 `java.util.Collections` 也是其中一员的，但我们不得不像下面这样使用他们：

```
//java
Collections.swap(list, Collections.binarySearch(list, Collection
s.max(otherList)), Collections.max(list))
```

由于这些类名总是不变的。我们可以使用静态导入并这样使用：

```
swap(list, binarySearch(list, max(otherList)), max(list))
```

这样就好很多了，但这样我们就只能从 IDE 自动完成代码那里获得很少或得不到帮助信息。如果我们可以像下面这样那么就好多了

```
list.swap(list.binarySearch(otherList.max()), list.max())
```

但我们又不想在 List 类中实现所有可能的方法。这就是扩展带来的好处。

数据类

我们经常创建一个只保存数据的类。在这样的类中一些函数只是机械的对它们持有的数据进行一些推导。在 kotlin 中这样的类称之为 data 类，用 `data` 标注:

```
data class User(val name: String, val age: Int)
```

编译器会自动根据主构造函数中声明的所有属性添加如下方法：

> `equals()` / `hashCode` 函数
>
> `toString` 格式是 "User(name=john, age=42)"
>
> [compontN()functions] (http://kotlinlang.org/docs/reference/multi-declarations.html) 对应按声明顺序出现的所有属性
>
> `copy()` 函数

如果在类中明确声明或从基类继承了这些方法，编译器不会自动生成。

为确保这些生成代码的一致性，并实现有意义的行为，数据类要满足下面的要求：

注意如果构造函数参数中没有 `val` 或者 `var` ，就不会在这些函数中出现；

> 主构造函数应该至少有一个参数；
>
> 主构造函数的所有参数必须标注为 `val` 或者 `var` ；
>
> 数据类不能是 abstract，open，sealed，或者 inner ；
>
> 数据类不能继承其它的类（但可以实现接口）。
>
> (在1.1之前)数据类只能实现接口。

从1.1开始数据类可以继承其它类（参考 Sealed classes ）

在 JVM 中如果构造函数是无参的，则所有的属性必须有默认的值，(参看Constructors);

> data class User(val name: String = "", val age: Int = 0)

复制

我们经常会对一些属性做修改但想要其他部分不变。这就是 `copy()` 函数的由来。在上面的 User 类中，实现起来应该是这样：

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

有了 copy 我们就可以像下面这样写了：

```
val jack = User(name = "jack", age = 1)
val olderJack = jack.copy(age = 2)
```

数据类和多重声明

组件函数允许数据类在多重声明中使用：

```
val jane = User("jane", 35)
val (name, age) = jane
println("$name, $age years of age") //打印出 "Jane, 35 years of age"
```

标准数据类

标准库提供了 `Pair` 和 `Triple` 。在大多数情形中，命名数据类是更好的设计选择，因为这样代码可读性更强而且提供了有意义的名字和属性。

泛型

像 java 一样，Kotlin 中的类可以拥有类型参数：

```
class Box<T>(t: T){
    var value = t
}
```

通常来说，创建一个这样类的实例，我们需要提供类型参数：

```
val box: Box<Int> = Box<Int>(1)
```

但如果类型有可能是推断的，比如来自构造函数的参数或者通过其它的一些方式，一个可以忽略类型的参数：

```
val box = Box(1)//1是 Int 型，因此编译器会推导出我们调用的是 Box<Int>
```

变型

java 类型系统最棘手的一部分就是通配符类型。但 kotlin 没有，代替它的是两种其它的东西：声明变型和类型投影(declaration-site variance and type projections)。

首先，我们想想为什么 java 需要这些神秘的通配符。这个问题在Effective Java,条目18中是这样解释的：使用界限通配符增加 API 的灵活性。首先 java 中的泛型是不变的，这就意味着 `List<String>` 不是 `List<Object>` 的子类型。为什么呢，如果 List 不是不变的，就会引发下面的问题：

```
// Java
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // !!! The cause of the upcoming problem sits here. Java prohibits this!
objs.add(1); // Here we put an Integer into a list of Strings
String s = strs.get(0); // !!! ClassCastException: Cannot cast Integer to String
```

因此 java 禁止了这样的事情来保证运行时安全。但这有些其它影响。比如，`Collection` 接口的 `addAll()` 方法。这个方法的签名在哪呢？直觉告诉我们应该是这样的：

```
//java
interface Collection<E> ... {
    void addAll(Collection<E> items);
}
```

但接下来我们就不能做下面这些操作了(虽然这些操作都是安全的)：

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
  to.addAll(from); // !!! Would not compile with the naive declaration of addAll:
                   //       Collection<String> is not a subtype of Collection<Object>
}
```

这就是为什么 `addAll()` 的签名是下面这样的：

```
//java
interface Collection<E> ... {
    void addAll(Colletion<? extend E> items);
}
```

这个通配符参数 `? extends T` 意味着这个方法接受一些 T 类型的子类而非 T 类型本身。这就是说我们可以安全的读 `T's` (这里表示 T 子类元素的集合)，但不能写，因为我们不知道 T 的子类究竟是什么样的，针对这样的限制，我们很想要这样的行为： `Collection<String>` 是 `Collection<? extens Object>` 的子类。换句话讲，带 **extends** 限定（上界）的通配符类型使得类型是协变的（**covariant**）。

这个技巧其实很简单：如果你只能从集合中读数据，那么使用 `String` 集合并从中读取 `Objects` 是安全的，如果你只能给 `存入` 集合 ，那么给 `Objects` 集合存入 `String` 也是可以的：在 Java 中 `List<? super String>` 是 `List<Object>` 的超类。

后者称为逆变性（**contravariance**），并且对于 `List <? super String>` 你只能调用接受 String 作为参数的方法 （例如，你可以调用 `add(String)` 或者 `set(int, String)` ），当然 如果调用函数返回 `List<T>` 中的 `T` ，你得到的并非一个 `String` 而是一个 `Object` 。

Joshua Bloch 称只能读取的对象为生产者，只能写入的对象为消费者。他建议："为了灵活性最大化，在表示生产者或消费者的输入参数上使用通配符类型"，并提出了以下助记符：

PECS 代表生产者-*Extens*，消费者-*Super*（*Producer-Extends, Consumer-Super*）。

注意：如果你使用一个生产者对象，如 `List<? extends Foo>` ，在该对象上不允许调用 `add()` 或 `set()` 。但这并不意味着 该对象是不可变的：例如，没有什么阻止你调用 `clear()` 从列表中删除所有项目，因为 `clear()` 根本无需任何参数。通配符（或其他类型的型变）保证的唯一的事情是类型安全。不可变性完全是另一回事。

声明处变型

假如有个范型接口 `Source<T>` ，没有任何接收 `T` 作为参数的方法，唯一的方法就是返回 `T` :

```
// Java
interface Source<T> {
  T nextT();
}
```

存储一个 `Source<String>` 的实例引用给一个类型为 `Source<Object>` 是十分安全的。但 Java并不知道，而且依然禁止这么做：

```
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // !!! Not allowed in Java
  // ...
}
```

为次，我们不得不声明对象类型为 `Source<? extends Object>` ，这样做并没有太大的意义，因为我们可以像以前一样调用所有方法，因此并没有通过复杂的类型添加什么值。但编译器不知道。

在 Kotlin 中，有种可以将这些东西解释给编译器的办法，叫做声明处变型：通过注解类型参数 `T` 的来源，来确保它仅从 `Source<T>` 成员中返回（生产），并从不被消费。 为此，我们提供 **out** 修饰符：

```
abstract class Source<out T> {
    abstract fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an
 out-parameter
    // ...
}
```

一般原则是：当一个类 `C` 的类型参数 `T` 被声明为 **out** 时，它就只能出现在 `C` 的成员的输出-位置，结果是 `C<Base>` 可以安全地作为 `C<Derived>` 的超类。

更聪明的说法就是，当类 C 在类型参数 T 之下是协变的，或者 T 是一个协变类型。可以把 C 想象成 T 的生产者，而不是 T 的消费者。

`out` 修饰符本来被称之为变型注解，但由于同处与类型参数声明处，我们称之为声明处变型。这与 Java 中的使用处变型相反。

另外除了 **out**，Kotlin 又补充了一个变型注释：**in**。它接受一个类型参数逆变：只可以被消费而不可以 被生产。非变型类的一个很好的例子是 `Comparable` ：

```
abstract class Comparable<in T> {
    abstract fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype
of Number
    // Thus, we can assign x to a variable of type Comparable<Do
uble>
    val y: Comparable<Double> = x // OK!
}
```

我们相信 **in** 和 **out** 两词是自解释的（因为它们已经在 C# 中成功使用很长时间了），因此上面提到的助记符不是真正需要的，并且可以将其改写为更高的目标：

存在性（**The Existential**）转变：消费者 **in**, 生产者 **out!** :-)

类型投影

使用处变型：类型投影

声明类型参数 T 为 *out* 很方便，而且可以避免在使用出子类型的麻烦，但有些类 不能 限制它只返回 `T` ，Array 就是一个例子：

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { /* ... */ }
    fun set(index: Int, value: T) { /* ... */ }
}
```

这个类既不能是协变的也不能是逆变的，这会在一定程度上降低灵活性。考虑下面的函数：

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

该函数作用是复制 array ，让我们来实际应用一下：

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any) // Error: expects (Array<Any>, Array<Any>)
```

这里我们又遇到了同样的问题 `Array<T>` 中的 `T` 是不可变型的，因此 `Array<Int>` 和 `Array<Any>` 互不为对方的子类，导致复制失败。为什么呢？应为复制可能会有不合适的操作，比如尝试写入，当我们尝试将 Int 写入 String 类型的 array 时候将会导致 `ClassCastException` 异常。

我们想做的就是确保 `copy()` 不会做类似的不合适的操作，为阻止向 `from` 写入，我们可以这样：

```
fun copy(from: Array<out Any>, to: Array<Any>) {
 // ...
}
```

这就是类型投影：这里的 `from` 不是一个简单的 array， 而是一个投影，我们只能调用那些返回类型参数 `T` 的方法，在这里意味着我们只能调用 `get()` 。这是我们处理调用处变型的方法，类似 Java 中 `Array<? extends Object>` ，但更简单。

当然也可以用 `in` 做投影：

```
fun fill(dest: Array<in String>, value: String) {
    // ...
}
```

`Array<in String>` 对应 Java 中的 `Array<? super String>` ， `fill()` 函数可以接受任何 `CharSequence` 类型或 `Object` 类型的 array 。

星投影

有时你对类型参数一无所知，但任然想安全的使用它。保险的方法就是定一个该范型的投影，每个该范型的正确实例都将是该投影的子类。

Kotlin 提供了一种星投影语法：

- For `Foo<out T>` , where `T` is a covariant type parameter with the upper bound `TUpper` , `Foo<*>` is equivalent to `Foo<out TUpper>` . It means that when the `T` is unknown you can safely *read* values of `TUpper` from `Foo<*>` .
- For `Foo<in T>` , where `T` is a contravariant type parameter, `Foo<*>` is equivalent to `Foo<in Nothing>` . It means there is nothing you can *write* to `Foo<*>` in a safe way when `T` is unknown.
- For `Foo<T>` , where `T` is an invariant type parameter with the upper bound `TUpper` , `Foo<*>` is equivalent to `Foo<out TUpper>` for reading values and to `Foo<in Nothing>` for writing values.

If a generic type has several type parameters each of them can be projected independently. For example, if the type is declared as `interface Function<in T, out U>` we can imagine the following star-projections:

- `Function<*, String>` means `Function<in Nothing, String>` ;
- `Function<Int, *>` means `Function<Int, out Any?>` ;
- `Function<*, *>` means `Function<in Nothing, out Any?>` .

Note: star-projections are very much like Java's raw types, but safe. (这部分暂未翻译)

范型函数

函数也可以像类一样有类型参数。类型参数在函数名之前：

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString() : String {  // extension function
    // ...
}
```

调用范型函数需要在函数名后面制定类型参数：

```
val l = singletonList<Int>(1)
```

范型约束

指定类型参数代替的类型集合可以用通过范型约束进行限制。

上界(upper bound)

最常用的类型约束是上界，在 Java 中对应 `extends` 关键字：

```
fun <T : Comparable<T>> sort(list: List<T>) {
    // ...
}
```

冒号后面指定的类型就是上界：只有 `Comparable<T>` 的子类型才可以取代 `T` 比如：

```
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype of Comparable<HashMap<Int, String>>
```

默认的上界是 `Any?` 。在尖括号内只能指定一个上界。如果要指定多种上界，需要用 **where** 语句指定：

```
fun <T> cloneWhenGreater(list: List<T>, threshold: T): List<T>
    where T : Comparable,
          T : Cloneable {
  return list.filter { it > threshold }.map { it.clone() }
}
```

嵌套类

类可以嵌套在其他类中

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() //==2
```

内部类

类可以标记为 inner 这样就可以访问外部类的成员。内部类拥有外部类的一个对象引用：

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() //==1
```

参看这里了解更多 this 在内部类的用法

匿名内部类

匿名内部类的实例是通过 对象表达式 创建的：

```
window.addMouseListener(object: MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

如果对象是函数式的 java 接口的实例（比如只有一个抽象方法的 java 接口），你可以用一个带接口类型的 lambda 表达式创建它。

```
val listener = ActionListener { println("clicked") }
```

枚举类

枚举类最基本的用法就是实现类型安全的枚举

```
enum class Direction {
    NORTH,SOUTH,WEST
}
```

每个自举常量都是一个对象。枚举常量通过逗号分开。

初始化

因为每个枚举都是枚举类的一个实例，它们是可以初始化的。

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

匿名类

枚举实例也可以声明它们自己的匿名类

```
enum class ProtocolState {
    WAITING {
        override fun signal() = Taking
    },
    Taking{
        override fun signal() = WAITING
    };
    abstract fun signal(): ProtocolState
}
```

可以有对应的方法，以及复写基本方法。注意如果枚举定义了任何成员，你需要像在 java 中那样用分号 ; 把枚举常量定义和成员定义分开。

使用枚举常量

像 java 一样，Kotlin 中的枚举类有合成方法允许列出枚举常量的定义并且通过名字获得枚举常量。这些方法的签名就在下面列了出来(假设枚举类名字是 EnumClass)：

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

如果指定的名字在枚举类中没有任何匹配，那么 `valueOf()` 方法将会抛出参数异常。

每个枚举常量都有获取在枚举类中声明的名字和位置的方法：

```
name(): Sting
ordinal(): Int
```

枚举类也实现了 Comparable 接口，比较时使用的是它们在枚举类定义的自然顺序。

对象表达式和声明

有时候我们想要创建一个对当前类有一点小修改的对象，但不想重新声明一个子类。java 用匿名内部类的概念解决这个问题。Kotlin 用对象表达式和对象声明巧妙的实现了这一概念。

对象表达式

通过下面的方式可以创建继承自某种(或某些)匿名类的对象：

```
window.addMouseListener(object: MouseAdapter () {
    override fun mouseClicked(e: MouseEvent) {
        //...
    }
})
```

如果父类有构造函数，则必须传递相应的构造参数。多个父类可以用逗号隔开，跟在冒号后面：

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { ... }

val ab = object : A(1), B {
    override val y = 14
}
```

有时候我们只是需要一个没有父类的对象，我们可以这样写：

```
val adHoc = object {
    var x: Int = 0
    var y: Int = 0
}

print(adHoc.x + adHoc.y)
```

像 java 的匿名内部类一样，对象表达式可以访问闭合范围内的变量 (和 java 不一样的是，这些变量不用是 final 修饰的)

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0
    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }
        override fun mouseEntered(e: MouseEvent){
            enterCount++
        }
    })
}
```

对象声明

单例模式是一种很有用的模式，Kotln 中声明它很方便：

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

这叫做对象声明，跟在 object 关键字后面是对象名。和变量声明一样，对象声明并不是表达式，而且不能作为右值用在赋值语句。

想要访问这个类，直接通过名字来使用这个类：

```
DataProviderManager.registerDataProvider(...)
```

这样类型的对象可以有父类型：

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
}
```

注意：对象声明不可以是局部的(比如不可以直接在函数内部声明)，但可以在其它对象的声明或非内部类中进行内嵌入

伴随对象

在类声明内部可以用 companion 关键字标记对象声明：

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

伴随对象的成员可以通过类名做限定词直接使用：

```
val instance = MyClass.create()
```

在使用了 `companion` 关键字时，伴随对象的名字可以省略：

```
class MyClass {
    companion object {

    }
}
```

注意，尽管伴随对象的成员很像其它语言中的静态成员，但在运行时它们任然是真正对象的成员实例，比如可以实现接口：

```
interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}
```

如果你在 JVM 上使用 `@JvmStatic` 注解，你可以有多个伴随对象生成为真实的静态方法和属性。参看 java interoperabillity。

对象表达式和声明的区别

他俩之间只有一个特别重要的区别：

> 对象表达式在我们使用的地方立即初始化并执行的
>
> 对象声明是懒加载的，是在我们第一次访问时初始化的。
>
> 伴随对象是在对应的类加载时初始化的，和 Java 的静态初始是对应的。

代理

类代理

代理模式 给实现继承提供了很好的代替方式，Kotlin 在语法上支持这一点，所以并不需要什么样板代码。`Derived` 类可以继承 `Base` 接口并且指定一个对象代理它全部的公共方法：

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { printz(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

在 `Derived` 的父类列表中的 by 从句会将 `b` 存储在 `Derived` 内部对象，并且编译器会生成 `Base` 的所有方法并转给 `b` 。

代理属性

Kotlin 很多常用属性，虽然我们可以在每次需要的时候手动实现它们，但更好的办法是一次实现多次使用，并放到库里。比如：

> 延迟属性：只在第一次访问时计算它的值。 可观察属性：监听者从这获取这个属性更新的通知。 在 map 中存储的属性，而不是为每个属性创建一个字段。

为了满足这些情形，Kotllin 支持代理属性：

```
class Example {
    var p: String by Delegate()
}
```

语法结构是： `val/var <property name>: <Type> by <expression>` 在 by 后面的表达式就是代理，因为 `get()` `set()` 对应的属性会被 `getValue()` `setValue()` 方法代理。属性代理不需要任何接口的实现，但必须要提供 `getValue()` 函数(如果是 var 还需要 `setValue()`)。像这样：

```
class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name} in $thisRef.'")
    }
}
```

当我们从 `p` 也就 是一个 `Delegate` 实例进行读取操作时，会调用 `Delegate` 的 `getValue()` 函数，因此第一个参数是我们从 `p` 中读取的，第二个参数是持有 `p` 的一个描述。比如：

```
val e = Example()
println(e.p)
```

打印结果：

> Example@33a17727, thank you for delegating ‘p’ to me!

同样当我们给 `p` 赋值时 `setValue()` 函数就将被调用。前俩个参数是一样的，第三个持有分配的值：

```
e.p = "NEW"
```

打印结果：

> NEW has been assigned to ‘p’ in Example@33a17727.

从 Kotlin 1.1 开始支持在函数内部或者代码块内声明代理，而不必是类成员。

标准代理

Kotlin 标准库为几种常用的代理提供了工厂方法

延迟

`lazy()` 是一个接受 lamdba 并返回一个实现延迟属性的代理：第一次调用 `get()` 执行 lamdba 并传递 `lazy()` 并存储结果，以后每次调用 `get()` 时只是简单返回之前存储的值。

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
```

上面代码的输出是：

```
computed!
Hello
Hello
```

默认情况下延迟属性的计算是同步的：该值的计算只在一个线程里，其他所有线程都将读取同样的值。如果代理不需要同步初始化，而且允许出现多线程同时执行该操作，可以传 `LazyThreadSafetyMode.PUBLICATION` 参数给 `lazy()` 。如果你确信初始化只会在单线程中出现，那么可以使用 `LazyThreadSafetyMode.NONE` 该模式不会提供任何线程安全相关的保障。

如果你想要线程安全，使用 `blockingLazy()`：它还是按照同样的方式工作，但保证了它的值只会在一个线程中计算，并且所有的线程都获取的同一个值。

可观察属性

`Delegates.observable()` 需要两个参数：一个初始值和一个用于修改的 handler 。每次我们给属性赋值时都会调用handler (在初始赋值操作后)。它有三个参数：一个将被赋值的属性，旧值，新值：

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

打印结果

> \ -> first
>
> first -> second

如果你想能够打断赋值并取消它，用 `vetoable()` 代替 `observable()` 。传递给 `vetoable` 的 handler 会在赋新值之前调用。

在 Map 中存储属性

把属性值存储在 map 中是一种常见的使用方式，这种操作经常出现在解析 JSON 或者其它动态的操作中。这种情况下你可以使用 map 来代理它的属性。

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

在这个例子中，构造函数接受一个 map :

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

代理属性将从这个 map 中取指(通过属性的名字)：

```
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```

var 属性可以用 MutableMap 代替只读的 `Map` ：

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

本地代理属性（从1.1开始支持）

你可以声明本地变量作为代理属性。比如你可以创建一个本地延迟变量：

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

`memoizedFoo` 只会在第一次访问时求值。如果 `someCondition` 不符合，那么该变量将根本不会被计算。

属性代理的要求

这里总结一些代理对象的要求。

只读属性 (val)，代理必须提供一个名字叫 `getValue` 的函数并接受如下参数：

> `thisRef` 接收者--必须是属性拥有者是同一种类型，或者是其父类
>
> `property` 必须是 KProperty<*> 或者它的父类

这个函数必须返回同样的类型或子类作为属性。

可变属性 (var)，代理必须添加一个叫 `setValue` 的函数并接受如下参数：

> `thisRef` 与 `getValue()` 一样
>
> `property` 与 `getValue()` 一样
>
> 新值--必须和属性类型一致或是它的父类

`getValue()` 和 `setValue()` 函数必须作为代理类的成员函数或者扩展函数。扩展函数对与想要对对象代理原本没有的函数是十分有用。两种 函数必须标记 `operator` 关键字。

代理类可能实现 `ReadOnlyProperty` 和 `ReadWriteProperty` 中的一个并要求带有 `operator` 方法。这些接口在 Kotlin 标准库中有声明：

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, va
lue: T)
}
```

转换规则

在每个代理属性的实现的背后，Kotlin 编译器都会生成辅助属性并代理给它。 例如，对于属性 `prop`，生成隐藏属性 `prop$delegate`，而访问器的代码只是简单地代理给这个附加属性：

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler instead:
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::pr
op, value)
}
```

Kotlin 编译器在参数中提供了关于 `prop` 的所有必要信息：第一个参数 `this` 引用到类 `C` 外部的实例而 `this::prop` 是 `KProperty` 类型的反射对象，该对象描述 `prop` 自身。

注意，直接在代码中引用绑定的可调用引用的语法 `this::prop` 自 Kotlin 1.1 起才可用。

提供代理（自 1.1 起）

通过定义 `provideDelegate` 操作符，可以扩展创建属性实现所代理对象的逻辑。 如果 `by` 右侧所使用的对象将 `provideDelegate` 定义为成员或扩展函数，那么会调用该函数来创建属性代理实例。

`provideDelegate` 的一个可能的使用场景是在创建属性时检查属性一致性。

例如，如果你想要在绑定之前检查属性名称，可以这样写：

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // create delegate
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ...
 }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ...
 }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` 的参数与 `getValue` 相同：

- `thisRef` —— 必须与 属性所有者 类型（对于扩展属性——指被扩展的类型）相同或者是它的超类型，
- `property` —— 必须是类型 `KProperty<*>` 或其超类型。

在创建 `MyUI` 实例期间，为每个属性调用 `provideDelegate` 方法，并立即执行必要的验证。

如果没有这种打断属性与其代理之间的绑定的能力，为了实现相同的功能， 你必须显式传递属性名，这不是很方便：

```
// Checking the property name without "provideDelegate" function
ality
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
   checkProperty(this, propertyName)
   // create delegate
}
```

在生成的代码中， `provideDelegate` 方法用来初始化辅助 `prop$delegate` 属性的初始化。下面是属性声明 `val prop: Type by MyDelegate()` 生成的代码与 上面（当 `provideDelegate` 方法不存在时）生成的代码的对比：

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler
// when the 'provideDelegate' function is available:
class C {
    // calling "provideDelegate" to create the additional "deleg
ate" property
    private val prop$delegate = MyDelegate().provideDelegate(thi
s, this::prop)
    val prop: Type
        get() = prop$delegate.getValue(this, this::prop)
}
```

注意， `provideDelegate` 方法只影响辅助属性的创建，并不会影响为 getter 或 setter 生成的代码。

函数

函数声明

在 kotlin 中用关键字 `fun` 声明函数：

```
fun double(x: Int): Int {

}
```

函数用法

通过传统的方法调用函数

```
val result = double(2)
```

通过 `.` 调用成员函数

```
Sample().foo() // 创建Sample类的实例,调用foo方法
```

中缀符号

在满足以下条件时,函数也可以通过中缀符号进行调用:

> 它们是成员函数或者是扩展函数 只有一个参数 使用 `infix` 关键词进行标记

```
//给 Int 定义一个扩展方法
infix fun Int.shl(x: Int): Int {
...
}

1 shl 2 //用中缀注解调用扩展函数

1.shl(2)
```

参数

函数参数是用 Pascal 符号定义的 name:type。参数之间用逗号隔开，每个参数必须指明类型。

```
fun powerOf(number: Int, exponent: Int) {
...
}
```

默认参数

函数参数可以设置默认值,当参数被忽略时会使用默认值。这样相比其他语言可以减少重载。

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size ) {
...
}
```

默认值可以通过在type类型后使用 `=` 号进行赋值

命名参数

在调用函数时可以参数可以命名。这对于那种有大量参数的函数是很方便的.

下面是一个例子：

```
fun reformat(str: String, normalizeCase: Boolean = true,upperCas
eFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
...
}
```

我们可以使用默认参数

> reformat(str)

然而当调用非默认参数是就需要像下面这样：

```
reformat(str, true, true, false, '_')
```

使用命名参数我们可以让代码可读性更强：

```
reformat(str,
    normalizeCase = true,
    uppercaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
  )
```

如果不需要全部参数的话可以这样：

```
reformat(str, wordSeparator = '_')
```

注意,命名参数语法不能够被用于调用Java函数中,因为Java的字节码不能确保方法参数命名的不变性

不带返回值的参数

如果函数不会返回任何有用值，那么他的返回类型就是 `Unit` . `Unit` 是一个只有唯一值 `Unit` 的类型.这个值并不需要被直接返回:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

`Unit` 返回值也可以省略，比如下面这样：

```
fun printHello(name: String?) {
    ...
}
```

单表达式函数

当函数只返回单个表达式时，大括号可以省略并在 = 后面定义函数体

```
fun double(x: Int): Int = x*2
```

在编译器可以推断出返回值类型的时候,返回值的类型可以省略:

```
fun double(x: Int) = x * 2
```

明确返回类型

下面的例子中必须有明确返回类型,除非他是返回 `Unit` 类型的值,Kotlin 并不会对函数体重的返回类型进行推断,因为函数体中可能有复杂的控制流,他的返回类型未必对读者可见(甚至对编译器而言也有可能是不可见的)：

变长参数

函数的参数(通常是最后一个参数)可以用 vararg 修饰符进行标记：

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts)
        result.add(t)
    return result
}
```

标记后,允许给函数传递可变长度的参数：

```
val list = asList(1, 2, 3)
```

只有一个参数可以被标注为 `vararg` 。加入 `vararg` 并不是列表中的最后一个参数,那么后面的参数需要通过命名参数语法进行传值,再或者如果这个参数是函数类型,就需要通过lambda法则.

当调用变长参数的函数时，我们可以一个一个的传递参数，比如 `asList(1, 2, 3)` ，或者我们要传递一个 array 的内容给函数，我们就可以使用 * 前缀操作符：

```
val a = array(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

函数范围

Kotlin 中可以在文件顶级声明函数，这就意味者你不用像在Java,C#或是Scala一样创建一个类来持有函数。除了顶级函数，Kotlin 函数可以声明为局部的，作为成员函数或扩展函数。

局部函数

Kotlin 支持局部函数，比如在一个函数包含另一函数。

```
fun dfs(graph: Graph) {
  fun dfs(current: Vertex, visited: Set<Vertex>) {
    if (!visited.add(current)) return
    for (v in current.neighbors)
      dfs(v, visited)
  }

  dfs(graph.vertices[0], HashSet())
}
```

局部函数可以访问外部函数的局部变量(比如闭包)

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }
    dfs(graph.vertices[0])
}
```

局部函数甚至可以返回到外部函数 qualified return expressions

```
fun reachable(from: Vertex, to: Vertex): Boolean {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (current == to) return@reachable true
        if (!visited.add(current)) return
        for (v  in current.neighbors)
            dfs(v)
    }
    dfs(from)
    return false
}
```

成员函数

成员函数是定义在一个类或对象里边的

```
class Sample() {
    fun foo() { print("Foo") }
}
```

成员函数可以用 . 的方式调用

```
Sample.foo()
```

更多请参看类和继承

泛型函数

函数可以有泛型参数，样式是在函数名前加上尖括号。

```
fun <T> sigletonArray(item: T): Array<T> {
    return Array<T>(1, {item})
}
```

更多请参看泛型

内联函数

参看这里

扩展函数

参看这里

高阶函数和 lambda 表达式

参看这里

尾递归函数

Kotlin 支持函数式编程的尾递归。这个允许一些算法可以通过循环而不是递归解决问题，从而避免了栈溢出。当函数被标记为 `tailrec` 时，编译器会优化递归，并用高效迅速的循环代替它。

```
tailrec fun findFixPoint(x: Double = 1.0): Double
    = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
```

这段代码计算的是数学上的余弦不动点。Math.cos 从 1.0 开始不断重复，直到值不变为止，结果是 0.7390851332151607 这段代码和下面的是等效的：

```
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if ( x == y ) return y
        x = y
    }
}
```

使用 `tailrec` 修饰符必须在最后一个操作中调用自己。在递归调用代码后面是不允许有其它代码的，并且也不可以在 try/catch/finall 块中进行使用。当前的尾递归只在 JVM 的后端中可以用

高阶函数与 lambda 表达式

高阶函数

高阶函数就是可以接受函数作为参数或者返回一个函数的函数。比如 `lock()` 就是一个很好的例子，它接收一个 lock 对象和一个函数，运行函数并释放 lock;

```
fun <T> lock(lock: Lock, body: () -> T ) : T {
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

现在解释一下上面的代码吧：`body` 有一个函数类型 `() -> T`,把它设想为没有参数并返回 T 类型的函数。它引发了内部的 try 函数块，并被 `lock` 保护，结果是通过 `lock()` 函数返回的。

如果我们想调用 `lock()` ，函数，我们可以传给它另一个函数做参数，参看函数引用：

```
fun toBeSynchroized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchroized)
```

其实最方便的办法是传递一个字面函数(通常是 lambda 表达式)：

```
val result = lock(lock, {
sharedResource.operation() })
```

字面函数经常描述有更多细节，但为了继续本节，我们看一下更简单的预览吧：

> 字面函数被包在大括号里
>
> 参数在 `->` 前面声明(参数类型可以省略)
>
> 函数体在 `->` 之后

在 kotlin 中有一个约定，如果某一个函数的最后一个参数是函数，并且你向那个位置传递了一个 lambda 表达式，那么，你可以在括号外面定义这个 lambda 表达式：

```
lock (lock) {
    sharedResource.operation()
}
```

最后一个高阶函数的例子是 `map()` (of MapReduce):

```
fun <T, R> List<T>.map(transform: (T) -> R):
List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

函数可以通过下面的方式调用

```
val doubled = ints.map {it -> it * 2}
```

如果字面函数只有一个参数，则声明可以省略，名字就是 `it` ：

```
ints.map {it * 2}
```

这样就可以写LINQ-风格的代码了：

```
strings.filter{ it.length == 5 }.sortedBy{ it }.map{ it.toUpperCase() }
```

内联函数

有些时候可以用 内联函数 提高高阶函数的性能。

字面函数和函数表达式

字面函数或函数表达式就是一个"匿名函数"，也就是没有声明的函数，但立即作为表达式传递下去。想想下面的例子：

```
max(strings, {a, b -> a.length < b.length })
```

`max` 函数就是一个高阶函数,它接受函数作为第二个参数。第二个参数是一个表达式所以本生就是一个函数，即字面函数。作为一个函数，相当于：

```
fun compare(a: String, b: String) : Boolean = a.length < b.lengt
h
```

函数类型

一个函数要接受另一个函数作为参数，我们得给它指定一个类型。比如上面的 `max` 定义是这样的：

```
fun max<T>(collection: Collection<out T>, less: (T, T) -> Boolean
): T? {
    var max: T? = null
    for (it in collection)
        if (max == null || less(max!!, it))
            max = it
    return max
}
```

参数 `less` 是 `(T, T) -> Boolean` 类型，也就是接受俩个 `T` 类型参数返回一个 `Boolean` :如果第一个参数小于第二个则返回真。

在函数体第四行， `less` 是用作函数

一个函数类型可以像上面那样写，也可有命名参数，更多参看命名参数

```
val compare: (x: T,y: T) -> Int = ...
```

函数文本语法

函数文本的完全写法是下面这样的：

```
val sum = {x: Int,y: Int -> x + y}
```

函数文本总是在大括号里包裹着，在完全语法中参数声明是在括号内，类型注解是可选的，函数体是在 `->` 之后，像下面这样：

```
val sum: (Int, Int) -> Int = {x, y -> x+y }
```

函数文本有时只有一个参数。如果 kotlin 可以从它本生计算出签名，那么可以省略这个唯一的参数，并会通过 `it` 隐式的声明它：

```
ints.filter {it > 0}//这是 (it: Int) -> Boolean  的字面意思
```

注意如果一个函数接受另一个函数做为最后一个参数，该函数文本参数可以在括号内的参数列表外的传递。参看 callSuffix

函数表达式

上面没有讲到可以指定返回值的函数。在大多数情形中，这是不必要的，因为返回值是可以自动推断的。然而，如果你需要自己指定，可以用函数表达式来做：

```
fun(x: Int, y: Int ): Int = x + y
```

函数表达式很像普通的函数声明，除了省略了函数名。它的函数体可以是一个表达式(像上面那样)或者是一个块：

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

参数以及返回值和普通函数是一样的，如果它们可以从上下文推断出参数类型，则参数可以省略：

```
ints.filter(fun(item) = item > 0)
```

返回值类型的推导和普通函数一样：函数返回值是通过表达式自动推断并被明确声明

注意函数表达式的参数总是在括号里传递的。 The shorthand syntax allowing to leave the function outside the parentheses works only for function literals.

字面函数和表达式函数的另一个区别是没有本地返回。没有 lable 的返回总是返回到 fun 关键字所声明的地方。这意味着字面函数内的返回会返回到一个闭合函数，而表达式函数会返回到函数表达式自身。

闭包

一个字面函数或者表达式函数可以访问闭包，即访问自身范围外的声明的变量。不像 java 那样在闭包中的变量可以被捕获修改：

```
var sum = 0

ints.filter{it > 0}.forEach {
    sum += it
}
print(sum)
```

函数表达式扩展

除了普通的功能，kotlin 支持扩展函数。这种方式对于字面函数和表达式函数都是适用的。它们最重要的使用是在 Type-safe Groovy-style builders。

表达式函数的扩展和普通的区别是它有接收类型的规范。

```
val sum = fun Int.(other: Int): Int = this + other
```

接收类型必须在表达式函数中明确指定，但字面函数不用。字面函数可以作为扩展函数表达式，但只有接收类型可以通过上下文推断出来。

表达式函数的扩展类型是一个带接收者的函数：

```
sum : Int.(other: Int) -> Int
```

可以用 `.` 来使用这样的函数：

```
1.sum(2)
```

内联函数

使用高阶函数带来了相应的运行时麻烦：每个函数都是一个对象，它捕获闭包，即这些变量可以在函数体内被访问。内存的分配，虚拟调用的运行都会带来开销

但在大多数这种开销是可以通过内联文本函数避免。下面就是一个很好的例子。 `lock()` 函数可以很容易的在内联点调用。思考一下下面的例子：

```
lock(i) { foo() }
```

(Instead of creating a function object for the parameter and generating a call)，编译器可以忽略下面的代码：

```
lock.lock()
try {
    foo()
}
finally {
    lock.lock()
}
```

这不正是我们最开始想要的吗?

为了让编译器这样做，我们需要用 `inline` 标记 `lock()` 函数：

```
inline fun lock<T>(lock: Lock,body: ()-> T): T {
    //...
}
```

`inline` 标记即影响函数本身也影响传递进来的 lambda 函数：所有的这些都将被关联到调用点。

内联可能会引起生成代码增长，但我们可以合理的解决它(不要内联太大的函数)

@noinline

如果只需要在内联函数中内联部分Lambda表达式，可以使用 `@noinline` 注解来标记不需要内联的参数：

```
inline fun foo(inlined: () -> Uint, @noinline notInlined: () ->
Unit) {
    //...
}
```

内联的 lambda 只能在内联函数中调用，或者作为内联参数，但 `@noinline` 标记的可以通过任何我们喜欢的方式操控：存储在字段，(passed around etc)

注意如果内联函数没有内联的函数参数并且没有具体类型的参数，编译器会报警告，这样内联函数就没有什么优点的(如果你认为内联是必须的你可以忽略警告)

返回到非局部

在 kotlin 中，我们可以不加条件的使用 `return` 去退出一个命名函数或表达式函数。这意味这退出一个 lambda 函数，我们不得不使用标签，而且空白的 `return` 在 lambda 函数中是禁止的，因为 lambda 函数不可以造一个闭合函数返回：

```
fun foo() {
    ordinaryFunction {
        return // 错误  不可以在这返回
    }
}
```

但如果 lambda 函数是内联传递的，则返回也是可以内联的，因此允许下面这样：

```
fun foo() {
    inlineFunction {
        return //
    ]
}
```

注意有些内联函数可以调用传递进来的 lambda 函数，但不是在函数体，而是在另一个执行的上下文中，比如局部对象或者一个嵌套函数。在这样的情形中，非局部的控制流也不允许在lambda 函数中。为了表明，lambda 参数需要有

`InlineOptions.ONLY_LOCAL_RETURN` 注解：

```
inline fun f(inlineOptions(InlineOption.ONLY_LOCAL_RETURN) body:
() -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

内联 lambda 不允许用 break 或 continue ，但在以后的版本可能会支持。

实例化参数类型

有时候我们需要访问传递过来的类型作为参数：

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p?.parent
    }
    @suppress("UNCHECKED_CAST")
    return p as T
}
```

现在，我们创立了一颗树，并用反射检查它是否是某个特定类型。一切看起来很好，但调用点就很繁琐了：

```
myTree.findParentOfType(javaClass<MyTreeNodeType>() )
```

我们想要的仅仅是给这个函数传递一个类型，即像下面这样：

```
myTree.findParentOfType<MyTreeNodeType>()
```

为了达到这个目的，内联函数支持具体化的类型参数，因此我们可以写成这样：

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p?.parent
    }
    return p as T
}
```

我们用 refied 修饰符检查类型参数，既然它可以在函数内部访问了，也就基本上接近普通函数了。因为函数是内联的，所以不许要反射，像 `!is` ‘as‘这样的操作都可以使用。同时，我们也可以像上面那样调用它了 `myTree.findParentOfType<MyTreeNodeType>()`

尽管在很多情况下会使用反射，我们仍然可以使用实例化的类型参数 `javaClass()` 来访问它：

```
inline fun methodsOf<reified T>() = javaClass<T>().getMethods()

fun main(s: Array<String>) {
    println(methodsOf<String>().joinToString('\n'))
}
```

普通的函数(没有标记为内联的)不能有实例化参数。

更底层的解释请看spec document

协程(Coroutines)有些 APIs 是需要长时间运行，并且需要调用者阻塞直到这些调用完成（比如网络 IO ，文件 IO ，CPU 或者 GPU 比较集中的工作）。协程提供了一种避免线程阻塞并且用一种更轻量级，更易操控到操作：协程暂停。

协程把异步编程放入库中来简化这类操作。程序逻辑在协程中顺序表述，而底层的库会将其转换为异步操作。库会将相关的用户代码打包成回调，订阅相关事件，调度其执行到不同的线程（甚至不同的机器），而代码依然想顺序执行那么简单。

很多其它语言中的异步模型都可以用 Kotlin 协程实现为库。比如 C# ECMAScipt 中的 async/wait ，Go 语言中的 channels 和 `select` ，以及 C# 和 Python 的 generators/ `yield` 模型。下面的描述会详细解释提供这些结构的库。

阻塞和挂起

一般来说，协程是一种可以不阻塞线程但却可以被挂起的计算过程。线程阻塞总是昂贵的，尤其是在高负载的情形下，因为只有小部分的线程是实际运行的，因此阻塞它们会导致一些重要的任务被延迟。

而协程的挂起基本没有什么开销。没有上下文切换或者任何的操作系统的介入。最重要的是，挂起是可以背用户库控制的，库的作者可以决定在挂起时根据需要进行一些优化／日志记录／拦截等操作。

另一个不同就是协程不能被任意的操作挂起，而仅仅可以在被标记为 挂起点 的地方进行挂起。

挂起函数

当一个函数被 `suspend` 修饰时表示可以被挂起。

```
suspend fun doSomething(foo: Foo): Bar{
 ...
}
```

这样的函数被称为 挂起函数，因为调用它可能导致挂起协程（库可以在调用结果已经存在的情形下决定取消挂起）。挂起函数可以想正常函数那样接受参数返回结果，但只能在协程中调用或着被其他挂起函数调用。事实上启动一个协程至少需要一个挂起函数，而且常常时匿名的（比如lambda）。下面这个例子是一个简单的 `async()` 函数（来自 `kotlinx.coroutines` 库）：

```
fun <T> async(block: suspend() -> T)
```

这里的 `async()` 只是一个普通的函数（不是挂起函数），但 `block` 参数是一个带有 `suspend` 修饰的函数类型，所以当传递一个 lambda 给 `async()` 时，这会是一个挂起 lambda ，这样我们就可以在这里调用一个挂起函数了。

```
async{
  doSomething(foo)
}
```

继续类比， `await()` 函数可以是一个挂起函数(因此在 `await(){}` 语句块内仍然可以调用)，该函数会挂起协程直至指定操作完成并返回结果：

```
async {
    ...
    val result = computation.await()
    ...
}
```

更多关于 `async/await` 原理的内容请看这里

注意 `await()` 和 `doSomething()` 不能在像 main() 这样的普通函数中调用：

```
fun main(args: Array<String>) {
    doSomething() // 错误：挂起函数从非协程上下文调用
}
```

还有一点，挂起函数可以是虚函数，当覆写它们时，必须指定 suspend 修饰符：

```
interface Base {
    suspend fun foo()
}

class Derived: Base {
    override suspend fun foo() { …… }
}
```

`@RestrictsSuspension` 注解

扩展函数（以及lambda）可以被标记为 `suspend` 。这样方便了用户创建其他DSLs以及扩展其它API。有些情况下，库的作者需要阻止用户添加新的挂起线程的方案。

这时就需要 `@RestrictsSuspension` 注解了。当一个接收者类或者接口 `R` 被标注时，所有可挂起扩展都需要代理 `R` 的成员或者其它扩展。由于扩展时不能互相无限代理（会导致程序终止），这就保障了所有挂起都是通过调用 `R` 的成员，这样库作者就能完全掌控挂起方式了。

不过这样的场景不常见，它需要所有的挂起都通过库的特殊方式实现。比如，用下面的 `buildSequence()` 函数实现生成器时，必须保证协程中所有的挂起都是通过调用 `yield()` 或者 `yieldAll()` 来实现。这就是为什么 `SequenceBuilder` 被标注为 `@RestrictsSuspension` ：

```
@RestrictsSuspension
public abstract class SequenceBuilder<in T> {
    ...
}
```

可以参看Github 源码

协程内部机制

这里并不打算全盘解释协程内部的工作原理，而是给大家一个整体上的概念。

协程完全时通过编译技术（并不需要 VM 或者 OS 方面的支持）实现，挂起时借由代码转换实现。基本上所有的挂起函数（当然是有些优化措施，但这里我们不会深入说明）都被转换为状态机。在挂起前，下一个状态会存储在编译器生成的与本地变量关联的类中。到恢复协程时，本地变量会被恢复为挂起之前的状态。

挂起的协程可以存储以及作为一个对象进行传递，该协程会继续持有其状态和本地变量。这样的对象的类型时 `Continuation` ，代码转换的整体实现思路是基于经典的 Continuation-passing style 。所有挂起函数要有一个额外的参数类型 `Continuation` 。

更多的细节可以参看设计文档 。其它语言（比如C# ECMASript2016）中类似的 async/await 模型在这里都有描述，当然了其它语言的实现机制和 Kotlin 有所不同

协程的实验状态

协程的设计是实验性的，也就是说在后面的 releasees 版本中可能会有所变更。当在 Kotlin1.1 中编译协程时，默认会有警告：*The feature "coroutines" is experimental* 。可以通过 opt-in flag 来移除警告。

由于处于实验状态，协程相关的标准库都在 `kotlin.coroutines.experimental` 包下。当设计确定时实验状态将会取消，最后的API将会移到 `kotlin.coroutines` ,实验性的包将会保留（或许是作为一个单独的构建中）以保持兼容。

千万注意: 建议库作者可以采用同样的转换：为基于协程的 API 采用 "experimental" 前缀作包名（比如 `com.example.experimental` ）。当最终 API 发布时，遵循下面的步骤：

- 复制所有 API 到 `com.example` 包下
- 保留实验性大包做兼容。

这样可以减少用户的迁移问题。

标准 API

协程主要在三种层级中支持：

- 语言层面的支持（比如支持函数挂起）
- Kotlin 标准库中核心底层 API

- 可以直接在代码中使用的高级 API

底层 API：`kotlin.coroutines`

底层 API 比较少，强烈建议不要使用，除非要创建高级库。这部分 API 主要在两个包中：

- `kotlin.coroutines.experimental` 带有主要类型与下述原语
    - `createCoroutine()`
    - `startCoroutine()`
    - `suspendCoroutine()`
- `kotlin.coroutines.experimental.intrinsics` 带有更底层的内联函数如 `suspendCoroutineOrReturn`

关于这些 API 用法的更多细节可以在这里找到。

`kotlin.coroutines` 中的生成器API：

`kotlin.coroutines.experimental` 中唯一的“应用层面”的函数是：

- `buildSequence()`
- `buildIterator()`

这些和 `kotlin-stdlib` 打包在一起，因为和序列相关。事实上，这些函数（这里单独以 `buildSequence()` 作为事例）实现生成器提供了一种更加简单的构造延迟序列的方法：

```
val fibonacciSeq = buildSequence {
    var a = 0
    var b = 1

    yield(1)

    while (true) {
        yield(a + b)

        val tmp = a + b
        a = b
        b = tmp
    }
}
```

这里通过调用 `yield()` 函数生成新的斐波那契数，就可以生成一个无限的斐波那契数列。当遍历这样的数列时，每遍历一步就生成一个斐波那契数，这样就可以从中取出无限的斐波那契数。比如 `fibonacciSeq.take(8).toList()` 会返回 `[1, 1, 2, 3, 5, 8, 13, 21]` 。协程让这一实现开销更低。

为了演示正真的延迟序列，在 `buildSequence()` 中打印一些调试信息：

```
val lazySeq = buildSequence {
    print("START ")
    for (i in 1..5) {
        yield(i)
        print("STEP ")
    }
    print("END")
}

// Print the first three elements of the sequence
lazySeq.take(3).forEach { print("$it ") }
```

运行上面的代码运，如果我们输出前三个元素的数字与生成循环的 `STEP` 有交叉。这意味着计算确实是惰性的。要输出 `1` ，我们只执行到第一个 `yield(i)` ，并且过程中会输出 `START` 。然后，输出 `2` ，我们需要继续下一个

`yield(i)` ，并会输出 `STEP` 。 `3` 也一样。永远不会输出再下一个 `STEP` （以及 `END` ），因为我们没有请求序列的后续元素。

使用 `yieldAll()` 函数可以一次性生成序列所有值：

```
val lazySeq = buildSequence {
    yield(0)
    yieldAll(1..10)
}

lazySeq.forEach { print("$it ") }
```

`buildIterator()` 与 `buildSequence()` 作用相似，只不过返回值时延迟迭代器。

通过给 `SequenceBuilder` 类写挂起扩展，可以给 `buildSequence()` 添加自定义生成逻辑：

```
suspend fun SequenceBuilder<Int>.yieldIfOdd(x: Int) {
    if (x % 2 != 0) yield(x)
}

val lazySeq = buildSequence {
    for (i in 1..10) yieldIfOdd(i)
}
```

其它高级API： `kotlinx.coroutines`

Kotlin 标准库只提供与协程相关的核心 API 。主要有基于协程的库核心原语和接口可以使用。

大多数基于协程的应用程序级API都作为单独的库发布： `kotlinx.coroutines` 。这个库覆盖了

- 平台无关的异步编程此模块 `kotlinx-coroutines-core`
  - 包括类似 Go 语言的 `select` 和其他便利原语
  - 这个库的综合指南在这里查看。
- 基于 JDK 8 中的 `CompletableFuture` 的 API： `kotlinx-coroutines-jdk8`

- 基于 JDK 7 及更高版本 API 的非阻塞 IO（NIO）： `kotlinx-coroutines-nio`
- 支持 Swing (`kotlinx-coroutines-swing`) 和 JavaFx (`kotlinx-coroutines-javafx`)
- 支持 RxJava： `kotlinx-coroutines-rx`

这些库既提供了方便的 API ，也可以作为构建其它基于协程库的样板参考。

多重声明

有时候可以通过给对象插入多个成员函数做区别是很方便的，比如：

```
val (name, age) = person
```

这种语法叫多重声明。多重声明一次创建了多个变量。我们声明了俩个新变量：`name` `age` 并且可以独立使用：

```
println(name)
println(age)
```

多重声明被编译成下面的代码：

```
val name = persion.component1()
val age = persion.component2()
```

`component1()` `component2()` 是另一个转换原则的例子。任何类型都可以在多重分配的右边。当然了，也可以有 `component3()` `component4()` 等等

多重声明也可以在 for 循环中用

```
for ((a, b) in collection) { ... }
```

参数 `a` 和 `b` 是 `component1()` `component2()` 的返回值

例子：一个函数返回俩个值

要是一个函数想返回俩个值。比如，一个对象结果，一个是排序的状态。在 Kotlin 中的一个紧凑的方案是声明 data 类并返回实例：

```
data class Result(val result: Int, val status: Status)

fun function(...): Result {
    //...
    return Result(result, status)
}

val (result, status) = function(...)
```

数据类自动声明 `componentN()` 函数

注意：也可以使用标准类 `Pair` 并让函数返回 'Pair'，但可读性不是很强

例子：多重声明和 Map

转换 map 的最好办法可能是下面这样：

```
for ((key, value) in map) {

}
```

为了让这个可以工作，我们需要

> 通过提供 `iterator()` 函数序列化呈现 map 通过 `component1()` 和 `component1()` 函数是把元素成对呈现

事实上，标准库提供了这样的扩展：

```
fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entrySet().iterator()
fun <K, V> Map.Entry<K, V>.component1() = getKey()
fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

因此你可以用 for 循环方便的读取 map (或者其它数据集合)

Ranges

range 表达式是通过 rangeTo 函数形成的。rangeTo 函数拥有形如 .. 的操作符，该操作符是用 in 和 !in 实现的。 Range 可以对任何可比较的类型做操作，但对整数基本类型是优化过的。下面是些例子：

```
if (i in 1..10) {
    println(i)
}

if (x !in 1.0..3.0) println(x)

if (str in "island".."isle") println(str)
```

数字的范围有个附加的特性：它们可以迭代。编译器会把它转成类似于 java 的 for 循环的形式，且不用担心越界：

```
for (i in 1..4) print(i) // prints "1234"

for (i in 4..1) print(i) // prints nothing

for (x in 1.0..2.0) print("$x ") // prints "1.0 2.0 "
```

如果你想迭代数字并想反过来，这个相当简单，你可以使用 `downTo()` 函数

```
for (i in 4 downTo 1) print(i)
```

也可以使用指定步数的迭代，这个用到 `step()`

```
for (i in 1..4 step 2) print(i) // prints "13"

for (i in 4 downTo 1 step 2) print(i) // prints "42"

for (i in 1.0..2.0 step 0.3) print("$i ") // prints "1.0 1.3 1.6
 1.9 "
```

工作原理

在标准库中有俩种接口：Range 和 Progression

Range 表示数学范围上的一个间隔。它有俩个端点：start 和 end 。主要的操作符是 contains 通常在 in/!in 操作符内：

Progression 表示一个算数级数。它有一个 start 和 end 以及一个非零 increment 。Progression 是Iterable 的一个子类，因此可以使用在 for 循环中，或者 map filter 等等。第一个元素是 start 下一个元素都是前一个元素的 increment 。`Progression` 的迭代与 java/javaScript 的 for 循环相同：

```
// if increment > 0
for (int i = start; i <= end; i += increment) {
  // ...
}
// if increment < 0
for (int i = start; i >= end; i += increment) {
  // ...
}
```

范围指标

使用例子：

```
// Checking if value of comparable is in range. Optimized for nu
mber primitives.
if (i in 1..10) println(i)

if (x in 1.0..3.0) println(x)

if (str in "island".."isle") println(str)

// Iterating over arithmetical progression of numbers. Optimized
 for number primitives (as indexed for-loop in Java).
for (i in 1..4) print(i) // prints "1234"

for (i in 4..1) print(i) // prints nothing

for (i in 4 downTo 1) print(i) // prints "4321"

for (i in 1..4 step 2) print(i) // prints "13"

for (i in (1..4).reversed()) print(i) // prints "4321"

for (i in (1..4).reversed() step 2) print(i) // prints "42"

for (i in 4 downTo 1 step 2) print(i) // prints "42"

for (x in 1.0..2.0) print("$x ") // prints "1.0 2.0 "

for (x in 1.0..2.0 step 0.3) print("$x ") // prints "1.0 1.3 1.6
 1.9 "

for (x in 2.0 downTo 1.0 step 0.3) print("$x ") // prints "2.0 1
.7 1.4 1.1 "

for (str in "island".."isle") println(str) // error: string rang
e cannot be iterated over
```

常见的接口的定义

有俩种基本接口： `Range` `Progression`

`Range` 接口定义了一个范围，或者是数学意义上的一个间隔。

```
interface Range<T : Comparable<T>> {
    val start: T
    val end: T
    fun contains(Element : T): Boolean
}
```

`Progression` 定义了数学上的级数。包括 start end increment 端点。最大的特点就是它可以迭代，因此它是 Iterable 的子类。end 不是必须的。

```
interface Progression<N : Number> : Iterable<N> {
    val start : N
    val end : N
    val increment : Number
}
```

与 java 的 for 循环类似：

```
// if increment > 0
for (int i = start; i <= end; i += increment) {
  // ...
}

// if increment < 0
for (int i = start; i >= end; i += increment) {
  // ...
}
```

类的实现

为避免不需要的重复，让我们先考虑一个数字类型 `Int` 。其它的数字类型也一样。注意这些类的实例需要用相应的构造函数来创建，使用 rangeTo() downTo() reversed() stop() 实用函数。

IntProgression 类很直接也很简单：

```
class IntProgression(override val start: Int, override val end: Int, override val increment: Int ): Progression<Int> {
    override fun iterator(): Iterator<Int> = IntProgressionIteratorImpl(start, end, increment)
}
```

`IntRange` 有些狡猾：它实现了 `Progression<Int>` `Range<Int>` 接口，因为它天生以通过 range 迭代(默认增加值是 1)：

```
class IntRange(override val start: Int, override val end: Int): Range<Int>, Progression<Int> {
  override val increment: Int
    get() = 1
  override fun contains(element: Int): Boolean = start <= element && element <= end
  override fun iterator(): Iterator<Int> = IntProgressionIteratorImpl(start, end, increment)
}
```

`ComparableRange` 也很简单：

```
class ComparableRange<T : Comparable<T>>(override val start: T, override val end: T): Range<T> {
  override fun contains(element: T): Boolean = start <= element && element <= end
}
```

一些实用的函数

rangeTo()

`rangeTo()` 函数仅仅是调用 *Range 的构造函数，比如：

```
class Int {
    fun rangeTo(other: Byte): IntRange = IntRange(this, Other)
    fun rangeTo(other: Int): IntRange = IntRange(this, other)
}
```

downTo()

`downTo()` 扩展函数可以为任何数字类型定义，这里有俩个例子：

```
fun Long.downTo(other: Double): DoubleProgression {
    return DoubleProgression(this, other, -1.0)
}

fun Byte.downTo(other: Int): IntProgression {
    return IntProgression(this, other, -1)
}
```

reversed()

`reversed()` 扩展函数是给所有的 `*Range` 和 `*Progression` 类定义的，并且它们都返回反向的级数。

```
fun IntProgression.reversed(): IntProgression {
    return IntProgression(end, start, -increment)
}

fun IntRange.reversed(): IntProgression {
    return IntProgression(end, start, -1)
}
```

step()

`step()` 扩展函数是给所有的 `*Range` 和 `*Progression` 类定义的，所有的返回级数都修改了 step 值。注意 step 值总是正的，否则函数不会改变迭代的方向。

```
fun IntProgression.step(step: Int): IntProgression {
  if (step <= 0) throw IllegalArgumentException("Step must be po
sitive, was: $step")
  return IntProgression(start, end, if (increment > 0) step else
 -step)
}

fun IntRange.step(step: Int): IntProgression {
  if (step <= 0) throw IllegalArgumentException("Step must be po
sitive, was: $step")
  return IntProgression(start, end, step)
}
```

类型检查和转换

is !is 表达式

我们可以在运行是通过上面俩个操作符检查一个对象是否是某个特定类：

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // same as !(obj is String)
    print("Not a String")
}
else {
    print(obj.length)
}
```

智能转换

在很多情形中，需要使用非明确的类型，因为编译器会跟踪 `is` 检查静态变量，并在需要的时候自动插入安全转换：

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

编译器足够智能如何转换是安全的，如果不安全将会返回：

```
if (x !is String) return
print(x.length) //x 自动转换为 String
```

或者在 `||` `&&` 操作符的右边的值

```
 // x is automatically cast to string on the right-hand side of
`||`
  if (x !is String || x.length == 0) return

  // x is automatically cast to string on the right-hand side of
 `&&`
  if (x is String && x.length > 0)
      print(x.length) // x is automatically cast to String
```

这样的转换在 when 表达式和 whie 循环中也会发生

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is Array<Int> -> print(x.sum())
}
```

“不安全”的转换符和

如果转换是不被允许的那么转换符就会抛出一个异常。因此我们称之为不安全的。在kotlin 中　我们用前缀 as 操作符

```
val x: String = y as String
```

注意 null 不能被转换为 `String` 因为它不是 `nullable` ，也就是说如果 `y` 是空的，则上面的代码会抛出空异常。

为了 java 的转换语句匹配我们得像下面这样：

```
val x: String?= y as String?
```

"安全"转换符

为了避免抛出异常，可以用 as? 这个安全转换符，这样失败就会返回 null　：

```
val x: String ?= y as? String
```

不管 as? 右边的是不是一个非空 `String` 结果都会转换为可空的。

This 表达式

为了记录下当前接受者，我们使用 this 表达式：

> 在类的成员中，this 表示当前类的对象
>
> 在扩展函数或扩展字面函数中，this 表示 . 左边接收者参数

如果 this 没有应用者，则指向的是最内层的闭合范围。为了在其它范围中返回 this ，需要使用标签

this使用范围

为了在范围外部(一个类，或者表达式函数，或者带标签的扩展字面函数)访问 this ，我们需要在使用 `this@lable` 作为 lable

```
class A { // implicit label @A
  inner class B { // implicit label @B
    fun Int.foo() { // implicit label @foo
      val a = this@A // A's this
      val b = this@B // B's this

      val c = this // foo()'s receiver, an Int
      val c1 = this@foo // foo()'s receiver, an Int

      val funLit = @lambda {String.() ->
        val d = this // funLit's receiver
        val d1 = this@lambda // funLit's receiver
      }


      val funLit2 = { (s: String) ->
        // foo()'s receiver, since enclosing function literal
        // doesn't have any receiver
        val d1 = this
      }
    }
  }
}
```

相等

在 kotlin 中有俩中相等：

> 参照相等(指向相同的对象) 结构相等

参照相等

参照相等是通过 `===` 操作符判断的(不等是 `!==`) a===b 只有 a b 指向同一个对象是判别才成立。

另外，你可以使用内联函数 `identityEquals()` 判断参照相等：

```
a.identityEquals(b)
a identityEquals b
```

结构相等

结构相等是通过 `==` 判断的。像 `a == b` 将会翻译成：

```
a?.equals(b) ?: b === null
```

如果 a 不是 null 则调用 `equals(Any?)` 函数，否则检查 b 是否参照等于 null

注意完全没有必要为优化你的代码而将 `a == null` 写成 `a === null` 编译器会自动帮你做的。

运算符重载

Kotlin 允许我们实现一些我们自定义类型的运算符实现。这些运算符有固定的表示，和固定的优先级。为实现这样的运算符，我们提供了固定名字的数字函数和扩展函数，比如二元运算符的左值和一元运算符的参数类型。

转换

这里我们描述了一些常用运算符的重载

一元运算符

表达式	转换
+a	a.plus()
-a	a.minus()
!a	a.not()

这张表解释了当编译器运行时，比如，表达式 `+a` ，是这样运行的：

> 决定 a 的类型，假设是 T 寻找接收者是 T 的无参函数 `plus()` ，比如数字函数或者扩展函数 如果这样的函数缺失或不明确，则返回错误。 如果函数是当前函数或返回类型是 `R` 则表达式 `+a` 是 `R` 类型。

注意这些操作符和其它的一样，都被优化为基本类型并且不会产生多余的开销。

表达式	转换
a++	a.inc() + see below
a--	a.dec() + see below

这些操作符允许修改接收者和返回类型。

```
inc()/dec() shouldn’t mutate the receiver object.
By “changing the receiver” we mean the receiver-variable, not th
e receiver object.
```

编译器是这样解决有后缀的操作符的比如 `a++` ：

> 决定 a 的类型，假设是 T 寻找无参函数 `inc()` ，作用在接收者T 如果返回类型是 R ，则必须是 T 的子类

计算表达式的效果是：

> 把 a 的初始值存储在 a0 中 把 a.inc() 的结果作用在 a 上 把 a0 作为表达式的返回值

a-- 的步骤也是一样的

++a --a 的解决方式也是一样的

二元操作符

表达式	转换
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.mod(b)
a..b	a.rangeTo(b)

编译器只是解决了该表中翻译为列的表达式

表达式	转换
a in b	b.contains(a)
a !in b	!b.contains(a)

in 和 !in 的产生步骤是一样的，但参数顺序是相反的。

标志	转换
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ... , i_n)
a[i] = b	a.set(i, b)
a[i,j] =b	a.set(i, j, b)
a[i_1, ... , i_n] = b	a.set(i_1,... ,o_n,b)

方括号被转换为 get set 函数

标志	转换
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ... , i_n)	a.invoke(i_1, ..., i_n)

括号被转换为带有正确参数的 invoke 参数

表达式	转换
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.modAssign(b)

在分配 a+= b时编译器是下面这样实现的：

> 右边列的函数是否可用 对应的二元函数(比如 plus())是否也可用,不可用在报告错误 确定它的返回值是 `Unit` 否则报告错误 生成 `a.plusAssign(b)` 否则试着生成 a=a+b 代码

Note: assignments are NOT expressions in Kotlin.

表达式	转换
a == b	a?.equals(b) ?: b.identityEquals(null)
a != b	!(a?.equals(b) ?: b.identityEquals(null))

注意 === !== 是不允许重载的

== 操作符有俩点特别：

> 它被翻译成一个复杂的表达式，用于筛选空值，而且 null == null 是真
>
> 它需要带有特定签名的函数，而不仅仅是特定名称的函数，下面这样：

```
fun equals(other: Any?): Boolean
```

或者用相同的参数列表和返回类型的扩展功能

标志	转换

a > b | a.compareTo(b) > 0 a < b | a.compareTo(b) < 0 a >= b | a.compareTo(b) >= 0 a <= b | a.compareTo(b) <= 0

所有的比较都转换为 `compareTo` 的调用，这个函数需要返回 `Int` 值

命名函数的中缀调用

我们可以通过 中缀函数的调用 来模拟自定义中缀操作符

空安全

可空类型和非空类型

Kotlin 类型系统致力于消灭空引用。

在许多语言中都存在的一个大陷阱包括 java ，就是访问一个空引用的成员，结果会有空引用异常。在 java 中这就是 `NullPointerException` 或者叫 NPE

Kotlin 类型系统致力与消灭 `NullPointerException` 异常。唯一可能引起 NPE 异常的可能是：

> 明确调用 `throw NullPointerException()` 外部 java 代码引起 一些前后矛盾的初始化(在构造函数中没初始化的成员在其它地方使用)

在 Kotlin 类型系统中可以为空和不可为空的引用是不同的。比如，普通的 `String` 类型的变量不能为空：

```
var a: String ="abc"
a = null //编译错误
```

允许为空，我们必须把它声明为可空的变量：

```
var b: String? = "abc"
b = null
```

现在你可以调用 a 的方法，而不用担心 NPE 异常了：

```
val l = a.length()
```

但如果你想使用 b 调用同样的方法就有可能报错了：

```
val l = b.length() //错误：b 不可为空
```

但我们任然想要调用方法，有些办法可以解决。

在条件中检查 null

首先，你可以检查 `b` 是否为空，并且分开处理下面选项：

```
val l = if (b != null) b.length() else -1
```

编译器会跟踪你检查的信息并允许在 if 中调用 length()。更复杂的条件也是可以的：

```
if (b != null && b.length() >0)
  print("Stirng of length ${b.length}")
else
  print("Empty string")
```

注意只有在 b 是不可变时才可以

安全调用

第二个选择就是使用安全操作符，`?.`：

```
b?.length()
```

如果 b 不为空则返回长度，否则返回空。这个表达式的的类型是 Int?

安全调用在链式调用是是很有用的。比如，如果 Bob 是一个雇员可能分配部门(也可能不分配)，如果我们想获取 Bob 的部门名作为名字的前缀，就可以这样做：

```
bob?.department?.head?.name
```

这样的调用链在任何一个属性为空都会返回空。

Elvis 操作符

当我们有一个 r 的可空引用时，我们可以说如果 `r` 不空则使用它，否则使用使用非空的 x：

```
val l: Int = if (b != null) b.length() else -1
```

尽管使用 if 表达式我们也可以使用 Elvis 操作符，`?:`

```
val l = b.length()?: -1
```

如果 ?: 左边表达式不为空则返回，否则返回右边的表达式。注意右边的表带式只有在左边表达式为空是才会执行

注意在 Kotlin 中 throw return 是表达式，所以它们也可以在 Elvis 操作符右边。这是非常有用的，比如检查函数参数是否为空；

```
fun foo(node: Node): String? {
  val parent = node.getParent() ?: return null
  val name = node.getName() ?: throw IllegalArgumentException("n
ame expected")

  //...
}
```

!! 操作符

第三个选择是 NPE-lovers。我们可以用 b!! ，这会返回一个非空的 b 或者抛出一个 b 为空的 NPE

```
val l = b !!.length()
```

安全转换

普通的转换可能产生 `ClassCastException` 异常。另一个选择就是使用安全转换，如果不成功就返回空：

```
val aInt: Int? = a as? Int
```

异常

异常类

所有的异常类都是 `Exception` 的子类。每个异常都有一个消息，栈踪迹和可选的原因。

使用 throw 表达式，抛出异常

```
throw MyException("Hi There!")
```

使用 try 捕获异常

```
try {
  // some code
}
catch (e: SomeException) {
  // handler
}
finally {
  // optional finally block
}
```

有可能有不止一个的 catch 块。finally 块可以省略。

try 是一个表达式

try 可以有返回值：

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatExce
ption) { null }
```

try 返回值要么是 try 块的最后一个表达式，要么是 catch 块的最后一个表达式。finally 块的内容不会对表达式有任何影响。

检查异常

Kotlin 中没有异常检查。这是由多种原因造成的，我们这里举个简单的例子

下面是 JDK `StringBuilder` 类实现的一个接口

```
Appendable append(CharSequence csq) throws IOException;
```

这个签名说了什么？ 它说每次我把 string 添加到什么东西(StringBuilder 或者 log console 等等)上时都会捕获 `IOExceptions` 为什么呢？因为可能涉及到 IO 操作(Writer 也实现了 Appendable)... 所以导致所有实现 Appendable 的接口都得捕获异常

```
try {
  log.append(message)
}
catch (IOException e) {
  // Must be safe
}
```

这样是不利的，参看Effective java

Bruce Eckel 在java 需要异常检查吗?说到：

> Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.

java 互动

参看 Java Interoperability section

注解

注解声明

注解是一种将元数据附加到代码中的方法。声明注解需要在类前面使用 annotation 关键字：

```
annotation class fancy
```

用法

```
@fancy class Foo {
    @fancy fun baz(@fancy foo: Int): Int {
        return (@fancy 1)
    }
}
```

在多数情形中 @ 标识是可选的。只有在注解表达式或本地声明中才必须：

```
fancy class Foo {
    fancy fun baz(fancy foo: Int): Int {
        @fancy fun bar() { ... }
        return (@fancy 1)
    }
}
```

如果要给构造函数注解，就需要在构造函数声明时添加 constructor 关键字，并且需要在前面添加注解：

```
class Foo @inject constructor (dependency: MyDependency)
    //...
```

也可以注解属性访问者：

```
class Foo {
    var x: MyDependency?=null
        @inject set
}
```

构造函数

注解可以有带参数的构造函数。

```
annotation class special(val why: String)
special("example") class Foo {}
```

Lambdas

注解也可以用在 Lambda 中。这将会应用到 lambda 生成的 invoke() 方法。这对 Quasar框架很有用，在这个框架中注解被用来并发控制

```
annotation class Suspendable
val f = @Suspendable { Fiber.sleep(10) }
```

java 注解

java 注解在 kotlin 中是完全兼容的：

```
import org.junit.Test
import org.junit.Assert.*

class Tests {
  Test fun simple() {
    assertEquals(42, getTheAnswer())
  }
}
```

java 注解也可以通过在导入是重命名实现像修改者那样：

```
import org.junit.Test as test

class Tests {
  test fun simple() {
    ...
  }
}
```

因为 java 中注解参数顺序是没定义的，你不能通过传入参数的方法调用普通函数。相反，你需要使用命名参数语法：

```
//Java
public @interface Ann {
    int intValue();
    String stringValue(0;
}

//kotlin
Ann(intValue = 1, stringValue = "abc") class C
```

像 java 中那样，值参数是特殊的情形；它的值可以不用明确的名字。

```
public @interface AnnWithValue {
    String value();
}

//kotlin
AnnWithValue("abc") class C
```

如果java 中的 value 参数有数组类型，则在 kotlin 中变成 vararg 参数：

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
// Kotlin
AnnWithArrayValue("abc", "foo", "bar") class C
```

如果你需要明确一个类作为一个注解参数，使用 Kotlin 类KClass。Kotlin 编译器会自动把它转为 java 类，因此 java 代码就可以正常看到注解和参数了。

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out A
ny?>)

Ann(String::class, Int::class) class MyClass
```

注解实例的值在 kotlin 代码中是暴露属性。

```
// Java
public @interface Ann {
    int value();
}
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

反射

反射是一系列语言和库的特性，允许在运行是获取你代码结构。 Kotlin 把函数和属性作为语言的头等类，而且反射它们和使用函数式编程或反应是编程风格很像。

> On the Java platform, the runtime component required for using the reflection features is distributed as a separate JAR file (kotlin-reflect.jar). This is done to reduce the required size of the runtime library for applications that do not use reflection features. If you do use reflection, please make sure that the .jar file is added to the classpath of your project.

类引用

最基本的反射特性就是得到运行时的类引用。要获取引用并使之成为静态类可以使用字面类语法：

```
val c = MyClass::class
```

引用是一种 KClass类型的值。你可以使用 `KClass.properties` 和 `KClass.extensionProperties` 获取类和父类的所有属性引用的列表。

注意这与 java 类的引用是不一样的。参看 java interop section

函数引用

当有一个像下面这样的函数声明时：

```
fun isOdd(x: Int) =x % 2 !=0
```

我们可以通过 `isOdd(5)` 轻松调用，同样我们也可以把它作为一个值传递给其它函数。我们可以使用 `::` 操作符

```
val numbers = listOf(1, 2, 3)
println(numbers.filter( ::isOdd) ) //prints [1, 3]
```

这里 `::isOdd` 是是一个函数类型的值 `(Int) -> Boolean`

注意现在 `::` 操作符右边不能用语重载函数。将来，我们计划提供一个语法明确参数类型这样就可以使用明确的重载函数了。

如果需要使用一系列类，或者扩展函数，必须是需合格的，并且结果是扩展函数类型，比如。`String::toCharArray` 就带来一个 `String: String.() -> CharArray` 类型的扩展函数。

例子：函数组合

考虑一下下面的函数：

```
fun compose<A, B, C>(f: (B) -> C, g: (A) -> B): (A) -> C {
    return {x -> f(g(x))}
}
```

它返回一个由俩个传递进去的函数的组合。现在你可以把它用在可调用的引用上了：

```
fun length(s: String) = s.size
val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength)) // Prints "[a, abc]"
```

属性引用

在 kotlin 中访问顶级类的属性，我们也可以使用 `::` 操作符：

```
var x = 1
fun main(args: Array<String>) {
    println(::x.get())
    ::x.set(2)
    println(x)
}
```

`::x` 表达式评估为 `KProperty<Int>` 类型的属性，它允许我们使用 `get()` 读它的值或者使用名字取回它的属性。更多请参看docs on the KProperty class

对于可变的属性比如 `var y =1` , `::y` 返回类型为 `[KMutableProperty<Int>](http://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-mutable-property.html)` ，它有 `set()` 方法

访问一个类的属性成员，我们这样修饰：

```
class A(val p: Int)

fun main(args: Array<String>) {
    val prop = A::p
    println(prop.get(A(1))) // prints "1"
}
```

对于扩展属性：

```
val String.lastChar: Char
  get() = this[size - 1]

fun main(args: Array<String>) {
  println(String::lastChar.get("abc")) // prints "c"
}
```

与 java 反射调用

在 java 平台上，标准库包括反射类的扩展，提供了到 java 反射对象的映射(参看 kotlin.reflect.jvm 包)。比如，想找到一个备用字段或者 java getter 方法，你可以这样写：

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main(args: Array<String>) {
    println(A::p.javaGetter) // prints "public final int A.getP(
)"
    println(A::p.javaField)  // prints "private final int A.p"
}
```

构造函数引用

构造函数可以像方法或属性那样引用。只需要使用 `::` 操作符并加上类名。下面的函数是一个没有参数并且返回类型是 `Foo`：

```
calss Foo
fun function(factory : () -> Foo) {
    val x: Foo = factory()
}
```

我们可以像下面这样使用：

```
function(:: Foo)
```

动态类型

作为静态类型的语言，kotlin任然拥有与无类型或弱类型语言的调用，比如javaScript。为了方便使用， `dynamic` 应运而生：

```
val dyn: dynamic = ...
```

`dynamic` 类型关闭了 kotlin 的类型检查：

> 这样的类型可以分配任意变量或者在任意的地方作为参数传递 任何值都可以分配为 `dynamic` 类型，或者作为参数传递给任何接受 `dynamic` 类型参数的函数 这样的类型不做 null 检查

`dynamic` 最奇特的特性就是可以在 `dynamic` 变量上调用任何属性或任何方法： (The most peculiar feature of dynamic is that we are allowed to call any property or function with any parameters on a dynamic variable:)

```
dyn.whatever(1, "foo", dyn) // 'whatever' is not defined anywhere

dyn.whatever(*array(1, 2, 3))
```

在 javaScript 平台上这样的代码会按照现有的样子编译：Kotlin 中的 `dyn.whatever(1)` 在生成的代码中变成 JavaScript 语言的 `dyn.whatever(1)` 。

动态调用可以返回 `dynamic` 作为结果，因此我们可以轻松实现链式调用：

```
dyn.foo().bar.bat(0
```

当给动态调用传递一个 lambda 表达式时，所有的参数默认都是 `dynamic` ：

```
dyn.foo {
  x -> x.bar() // x is dynamic
}
```

更多细节参看spec document

Interop

交互

java 交互

Kotlin 在设计时就是以与 java 交互为中心的。现存的 java 代码可以在 kotlin 中使用，并且 Kotlin 代码也可以在 java 中流畅运行。这节我们会讨论在 kotlin 中调用 java 代码。

在 kotlin 中调用 java 代码

基本所有的 Java 代码都可以运行

```
import java.util.*
fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    for (item in source )
        list.add(item)
    for (i in 0..source.size() - 1)
        list[i] = source[i]
}
```

空的返回

如果 Java 方法返回空，则在 Kotlin 调用中返回 `Unit` 。如果

原文

待翻译 请暂时参考原文

使用Maven

插件和版本

Kotlin-maven-plugin 可以编译 Kotlin 资源和模块。现在只有 Maven V3 支持

通过 Kotlin.version 定义你想要的 Kotlin 版本。可以有以下的值

> X.Y.SNAPSHOT:对应版本 X.Y 的快照，在 CI 服务器上的每次成功构建的版本。这些版本不是真正的稳定版，只是推荐用来测试新编辑器的功能的。现在所有的构建都是作为 0.1-SNAPSHOT 发表的。你可以参看configure a snapshot repository in the pom file
>
> X.Y.X: 对应版本 X.Y.Z 的 release 或 milestone ，自动升级。它们是文件构建。Release 版本发布在 Maven Central 仓库。在 pom 文件里不需要多余的配置。

milestone 和 版本的对应关系如下：

Milestone	Version
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

配置快照仓库

使用 kotlin 版本的快照，需要在 pom 中这样定义：

```
<repositories>
    <repository>
        <id>sonatype.oss.snapshots</id>
        <name>Sonatype OSS Snapshot Repository</name>
        <url>http://oss.sonatype.org/content/repositories/snapsh
ots</url>
        <releases>
            <enabled>false</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
</repositories>

<pluginRepositories>
    <pluginRepository>
        <id>sonatype.oss.snapshots</id>
        <name>Sonatype OSS Snapshot Repository</name>
        <url>http://oss.sonatype.org/content/repositories/snapsh
ots</url>
    <releases>
        <enabled>false</enabled>
    </releases>
    <snapshots>
        <enabled>true</enabled>
    </snapshots>
    </pluginRepository>
</pluginRepositories>
```

依赖

kotlin 有一些扩展标准库可以使用。在 pom 文件中使用如下的依赖：

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

只编译 kotlin 源码

编译源码需要在源码文件夹打一个标签：

```
<sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirec
tory>
<testSourceDirectory>${project.basedir}/src/test/kotlin</testSou
rceDirectory>
```

在编译资源是需要引用kotlin Maven Plugin:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>
    <executions>
        <execution>
            <id>compile</id>
            <phase>compile</phase>
            <goals> <goal>compile</goal> </goals>
        </execution>
        <execution>
            <id>test-compile</id>
            <phase>test-compile</phase>
            <goals> <goal>test-compile</goal> </goals>
        </execution>
    </executions>
</plugin>
```

编译 kotlin 和 java 资源

为了编译混合代码的应用，Kotlin 编译器应该在 java 编译器之前先工作。在 maven 中意味着 kotlin-maven-plug 应该在 maven-compiler-plugin 之前。

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>0.1-SNAPSHOT</version>
    <executions>
        <execution>
            <id>compile</id>
            <phase>process-sources</phase>
            <goals> <goal>compile</goal> </goals>
        </execution>
        <execution>
            <id>test-compile</id>
            <phase>process-test-sources</phase>
            <goals> <goal>test-compile</goal> </goals>
        </execution>
    </executions>
</plugin>
```

使用扩展的注解

kotlin 使用扩展的注解解析 java 库的信息。为了明确这些注解，你需要像下面这样：

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>0.1-SNAPSHOT</version>
    <configuration>
        <annotationPaths>
            <annotationPath>path to annotations root</annotation
Path>
        </annotationPaths>
    </configuration>
```

例子

你可以在 Github 仓库参考

使用 Ant

获得 Ant 任务

Kotlin 提供了 Ant 三个任务:

> kotlinc : Kotlin 面向 JVM 的编译器
>
> kotlin2js: 面向 javaScript 的编译器
>
> withKotlin: 使用标准 javac Ant 任务时编译 Kotlin 文件的任务

这些任务定义学在 kotlin-ant.jar 标准库中，位与 kotlin compiler 的 lib 文件夹下

面向 JVM 的只有 kotlin 源文件任务

当项目只有 kotlin 源文件时，最简单的方法就是使用 kotlinc 任务：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" clas
spath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

${kotlin.lib} 指向 kotlin 单独编译器解压的文件夹

面向 JVM 的只有 kotlin 源文件但有多个根的任务

如果一个项目包含多个根源文件，使用 src 定义路径：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" clas
spath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
    </target>
</project>
```

面向 JVM 的有 kotlin 和 java 源文件

如果项目包含 java kotlin 代码，使用 kotlinc 是可以的，但建议使用 withKotlin 任务

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" clas
spath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false" srcdi
r="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

面向 JavaScript 的只有一个源文件夹的

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" clas
spath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

面向 JavaScript 有前缀，后缀以及 sourcemap 选项

```
<project name="Ant Task Test" default="build">
    <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml" clas
spath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="pre
fix" outputPostfix="postfix" sourcemap="true"/>
    </target>
</project>
```

##面向 JavaScript 只有一个源码文件夹并有元信息的选项

如果你想要描述 javaScript/Kotlin 库的转换结果， `mateInfo` 选项是很有用的。如果 `mateInfo` 设置为 true 则编译附加 javaScript 文件时会创建二进制的元数据。这个文件会与转换结果一起发布

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" clas
spath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <!-- out.meta.js will be created, which contains binary
descriptors -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
    </target>
</project>
```

参考

下面是所有的元素和属性

#kotlinc 属性名字|描述|必须性|默认值

---|---|---|---| src|要编译的Kotlin 文件或者文件夹|yes| output|目标文件夹或 .jar 文件名 |yes| classpath|类的完整路径|no| classpathref|类的完整路径参考|no| stdlib|"Kotlin-runtime.jar" 的完整路径|no|"" includeRuntime|如果输出是 .jar 文件，是否 kotlin 运行时库是否包括在 jar 中|no|true

#withKotlin 属性

名字	描述	必须性	默认值
src	要编译的Kotlin 文件或者文件夹	yes	
output	目标文件夹	yes	
library	库文件(kt,dir,jar)	no	
outputPrefix	生成 javaScript 文件的前缀	no	
outputSufix	生成 javaScript 文件的后缀	no	
sourcemap	是否生成 sourcemap	no	
metaInfo	是否生成二进制元数据文件描述	no	
main	是否生成调用主函数	no	

Griffon 支持参看provided externally

使用 Gradle

插件和版本

kotlin-gradle-plugin 可以编译 Kotlin 文件和模块

> X.Y.SNAPSHOT:对应版本 X.Y 的快照，在 CI 服务器上的每次成功构建的版本。这些版本不是真正的稳定版，只是推荐用来测试新编辑器的功能的。现在所有的构建都是作为 0.1-SNAPSHOT 发表的。你可以参看configure a snapshot repository in the pom file
>
> X.Y.X: 对应版本 X.Y.Z 的 release 或 milestone ，自动升级。它们是文件构建。Release 版本发布在 Maven Central 仓库。在 pom 文件里不需要多余的配置。

milestone 和 版本的对应关系如下：

Milestone	Version
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

面向 Jvm

对于 jvm，需要应用 kotlin 插件

> apply plugin: "kotlin"

至于 M11 ，kotlin 文件可以与 java 混用。默认使用不同文件夹：

```
project
    - src
        - main (root)
            - kotlin
            - java
```

如果不使用默认的设置则对应的文件属性要修改：

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

面向JavaScript

但目标是 JavaScript 时：

> apply plugin: "kotln2js"

这个插件只对 kotlin 文件起作用，因此建议把 kotlin 和 java 文件分开。对于 jvm 如果不用默认的值则需要修改源文件夹：

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}
```

如果你想建立一个复用的库，使用 `kotlinOptions.metaInfo` 生成附加的带附加二进制描述的 js 文件

```
compileKotlin2Js {
    kotlinOptions.metaInfo = true
}
```

目标是 android

Android Gradle 模块与普通的 Gradle 模块有些不同，所以如果你想建立 kotlin 写的android 项目，则需要下面这样：

```
buildscript {
    ...
}
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
```

Android Studio

如果使用 Android Studio,需要添加下面的代码：

```
android {
  ...

  sourceSets {
    main.java.srcDirs += 'src/main/kotlin'
  }
}
```

这是告诉 android studio kotlin 文件的目录位置方便 IDE 识别

配置依赖

我们需要添加 kotlin-gradle-plugin 和 kotlin 标准库依赖

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:<versio
n>'
  }
}

apply plugin: "kotlin" // or apply plugin: "kotlin2js" if target
ing JavaScript

repositories {
  mavenCentral()
}

dependencies {
  compile 'org.jetbrains.kotlin:kotlin-stdlib:<version>'
}
```

使用快照版本

如果使用快照版本则如下所示：

```
buildscript {
  repositories {
    mavenCentral()
    maven {
      url 'http://oss.sonatype.org/content/repositories/snapshot
s'
    }
  }
  dependencies {
    classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:0.1-SNA
PSHOT'
  }
}

apply plugin: "kotlin" // or apply plugin: "kotlin2js" if target
ing JavaScript

repositories {
  mavenCentral()
  maven {
    url 'http://oss.sonatype.org/content/repositories/snapshots'
  }
}

dependencies {
  compile 'org.jetbrains.kotlin:kotlin-stdlib:0.1-SNAPSHOT'
}
```

例子

Kotlin仓库有如下例子：

> Kotlin Mixed java and Kotlin Android javaScript

- 与 java 对比
- 与 Scala 对比

与 java 的对比

一些 java 的问题

Kotlin 修复了 java 的一系列问题

> Null 引用交给了类型系统控制
>
> 没有 raw 类型
>
> Arrays 在 kotlin 中是不变的
>
> kotlin 有合适的函数类型，作为 java SAM 转换的反对。(Kotlin has proper function types, as opposed to Java's SAM-conversions)
>
> Use-site variance without wildcards
>
> Kotlin 不强制检查异常

java 有的而 kotlin 没有

> 异常检查
>
> 原始类型不是类
>
> 静态成员
>
> 非私有成员
>
> 通配符类型

kotlin 有的而 java 没有

> 字面函数+内联函数=高性能自定义控制结构 扩展函数 空安全 智能转换 String 模板 性能 一级构造函数 First-class delegation 变量和属性类型的类型接口 单例模式 变量推断和类型预测 范围表达式 运算符重载 伴随对象

与 Scala 对比

Kotlin 设计时的俩个主要目标是：

> 至少和 java 运行速度一样快
>
> 在保证语言尽量简单的情况下在易用性上提高

考虑到这俩点，如果你喜欢 Scala ，你可能不需要 Kotlin

Scala 有而 Kotlin 没有的

> 隐式转换，隐式参数 --在 Scala 中，在不适用 debugger 的时候很难知道代码发生了什么，因为太多的东西是隐式的 --通过函数增加类型在 kotlin 中需要使用扩展函数
>
> 可重载和类型成员
>
> 路径依赖的类型
>
> 宏
>
> Existential types --类型推断是很特殊的情形
>
> 特征的初始化逻辑很复杂 --参看类和继承
>
> 自定义象征操作 --参看操作符重载
>
> 内建 xml --参看Type-safe Groovy-style builders

以后 kotlin可能会添加的特性：

> 结构类型
>
> 值类型
>
> Yield 操作符
>
> Actors
>
> 并行集合(Parallel collections)

Kotlin 有而 Scala 没有的

> 零开销的null安全
>
> - Scala的是Option，是在句法和运行时的包装
>
> Smart casts
>
> Kotlin 的内联函数非局部的跳转
>
> First-class delegation。也通过第三方插件：Autoproxy实现